ADDRESSING LOGICAL DEADLOCKS THROUGH TASK-PARALLEL LANGUAGE DESIGN

A Dissertation
Presented to
The Academic Faculty

By

Caleb Voss

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology

December 2020

ADDRESSING LOGICAL DEADLOCKS THROUGH TASK-PARALLEL LANGUAGE DESIGN

Approved by:

Dr. Vivek Sarkar, Advisor
School of Computer Science
Georgia Institute of Technology

Dr. Qirun Zhang
School of Computer Science
Georgia Institute of Technology

Dr. Alessandro Orso
School of Computer Science
Georgia Institute of Technology

Dr. Tiago Cogumbreiro
College of Science and Mathematics
University of Massachusetts Boston

Dr. David Devecsery
School of Computer Science
Georgia Institute of Technology

Date Approved: 5 November 2020

As you set out for Ithaka, hope that the journey is long...

Constantine Cavafy, "Ithaka"

In remembrance of Marie, who earned the first doctorate in the family;

and for my parents, who are my first and last advisors.

S. D. G.

ACKNOWLEDGMENTS

I owe my foremost thanks to my advisor, Dr. Vivek Sarkar. I met him in 2011 as an anxious high school junior when I sat in on his parallel programming lecture at Rice University. After the lecture he chased me down under the unforgettable Duncan Hall ceiling (those who have seen it will understand the memories of the ceiling are inseparable from the memories of conversations under the ceiling) to give me his contact information in case he could be of any help to me. Thank you, Vivek, for being in exactly the right place at exactly the moment I needed you, seven years later. Thank you for every meeting, every phone call, every piece of advice and direction, your critical eye, your encouragement, and your willingness to let me explore. Thank you for teaching me, for letting me teach with you, and for trusting me to teach without you.

To each member of my dissertation committee, Dr. Alex Orso, Dr. David Devecsery, Dr. Qirun Zhang, and Dr. Tiago Cogumbreiro, thank you for many hours of reading and listening, for your challenging questions, for your helpful suggestions, and for assisting me in ensuring my research is the best it can be.

I give many thanks to the National Science Foundation, which has made this dissertation possible through support under grants DGE-1650044 and CCF-1822919.

Ravi Mangal, thank you for letting me borrow your wisdom and your courage. Thank you for your ever sober-minded perspective, your realism, and your uncompromising idealism. I am not sure how you convinced me to read some of the densest literature in our field with no clear application to my research. I suppose it was

because we both just loved it.

To David Heath and Qi Zhou, my collaborators in program analysis, it is a great misfortune that none of the topics in this dissertation are related to our work together. I am so glad I was able to work with and along side you both for two years. Even if the chapter is missing from the dissertation, it is not missing from my graduate school journey, and it has been indispensable in setting my path forward.

Thank you to the entire Habanero research lab. I have certainly lost count, but my path has intersected with something like seventeen of you. Thank you for being so welcoming to me.

I still owe so much to Dr. Lydia Kavraki and Dr. Mark Moll, my undergraduate advisors. You expected so much from me but gave me so much more—not in return, but just because you cared. Mark, you taught me to be a researcher and a scientist. I learned more in your office than in any classroom. Lydia, you are the high standard of mentorship that I will always hold others to, unfairly for them. You opened my future wide for me and refused to settle for anything less than what you believed I could achieve. And I will have achieved much if I can challenge and inspire even one person as profoundly as you challenge and inspire every one of your students.

I reserve my most special thanks for my parents. You have so faithfully accompanied me through every up and every down. Thank you for giving me every shove I needed. Thank you for every bit of advice, solicited and not. Thank you for investing in me in whatever form that has taken, even when it has meant letting me roam thousands of miles away.

Finally, thank you to every friend who walked alongside me for any part of the last four and a half years, whether or not you knew that you were helping, or how. I cannot possibly list all of you (all y'all) or what you have meant to me. Thank you to every Bulldog who found room to accept this Yellow Jacket; to those of you who did not follow through on your threat to crash the classes I taught; to the friends who

reminded me, at times when I needed to hear it most, that the right choices for my career are not about maximizing my degree, that I do, in fact, have a career despite remaining a student for far too long, and that fulfillment in one's work cannot be measured in the end by its output. Thank you to the Fisks, for letting me stay far too late every time; to Juan Carlos Martinez, for a copy of *Every Good Endeavor*, which I saved until halfway through the program, when I knew I would need it most; and to the Agans, for the aptly timed reading of *Leaf by Niggle*, which was not, you may be glad to hear, an interruption. You have all helped bear the weight of this task when it was heavy, and you multiplied my joy in it when it was light. Thank you, my friends.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Task-parallel programming languages offer a variety of high-level mechanisms for synchronization that trade off between flexibility and deadlock safety. Certain approaches, such as spawn-sync and async-finish task organization, are deadlock-free by construction, but support limited synchronization patterns. However, more powerful approaches, such as promises and phasers, are trivial to deadlock. Viewing high-level task-parallel programming as the successor to low-level concurrent programming, it is imperative that language features offer both flexibility to avoid over-synchronization and also sufficient protection against logical deadlock bugs. Lack of flexibility leads to code that does not take full advantage of the available parallelism in the computation. Lack of deadlock protection leads to error-prone code in which a single bug can involve arbitrarily many tasks, making it difficult to reason about. We make advances in both flexibility and deadlock protection for existing task-parallel synchronization mechanisms by carefully designing dynamically verifiable usage policies and language constructs.

We first define a deadlock-freedom policy for *futures*. The rules of the policy follow naturally from the semantics of asynchronous task closures and correspond to a preorder traversal of the task tree. The policy admits an additional class of deadlock-free programs compared to past work. Each blocking wait for a future can be verified by a stateless, lock-free algorithm, resulting in low time and memory overheads at runtime.

In order to define and identify deadlocks for *promises*, we introduce a mecha-

nism for promises to be owned by tasks. Simple annotations make it possible to ensure that each promise is eventually fulfilled by the responsible task or handed off to another task. Ownership semantics allows us to formally define two kinds of promise bugs: omitted sets and deadlock cycles. We present novel detection algorithms for both bugs. We further introduce an approximate deadlock-freedom policy for promises that, instead of precisely detecting cycles, raises an alarm when synchronization dependences occurring between trees of tasks are at risk of deadlocking. To establish both the safety and the flexibility of the approach, we prove that this over-approximation safely identifies all deadlocks, and we prove that deadlock-free programs can be made to comply with the policy without loss of parallelism through the use of a novel language feature, the *guard* block, which acts as a hint to the verifier.

Finally, we identify a lack of flexibility in the *phaser*, a synchronization primitive that, under certain restrictions, is deadlock-free by construction. The traditional restrictions cause undesirable interaction between unrelated phasers and tasks, leading to poor program design and unnecessary synchronization. We extend the semantics of phasers by introducing the concept of a *subphase*. By organizing phasers and their phases more carefully, we can eliminate some over-synchronization and anti-modularity that occurs in phaser programs to recover performance while still enjoying deadlock freedom.

CHAPTER 1
INTRODUCTION

In a post-Moore computational world, software engineers are increasingly unable to rely on advancements in the speed of sequential computations to address performance requirements [23]. Instead, software solutions must take advantage of parallel hardware capabilities such as multi-core CPUs, distributed systems, and specialized accelerators [23, 44]. However, programming for a parallel system is prone to classes of software bugs that do not arise in sequential code, such as deadlocks and data races, which result from incorrect synchronization.

Programming language design has steadily assisted in solving concurrency bugs by pushing the software paradigm away from low-level concurrency primitives to the higher-level abstraction that is task-based parallelism [31, 17, 15]. This transition is akin to the shift from goto-based control flow to the block structure of if-then-else and explicit loops [26]. Such code is better organized, communicates intent, and can be reasoned about in a modular way. The rich constructs of task-based programming can be likewise thought of as higher-level parallel control flow mechanisms that are designed to make parallel code safer and easier to reason about. However, many of the predefined synchronization facilities that are provided by existing task-parallel languages can still lead to concurrency bugs when used improperly.

It remains an open problem in this domain to continue designing less error-prone synchronization facilities by introducing new constructs or imposing new usage policies on existing constructs. Ideally, code in task-parallel languages should approach

an almost algorithmic description of a program in which all its opportunities for parallelism are exposed, while the concrete implementation of a task-parallel program as a concurrent program on a real machine is left to the discretion of a runtime. The implications of this ideal are that task-parallel synchronization should be both highly expressive so that code is not over-synchronized and also restrictive so that concurrency bugs cannot manifest in this level of abstraction.

1.1 Task-parallel Programming

In a task-based programming paradigm, the program issues asynchronous tasks, which are instances of code that are eligible to be run in parallel. Examples of such languages and libraries include Cilk [31], X10 [17], the Habanero family [15, 50, 41], Scala [42], Chapel [16], Fortress [4], OpenMP [66], and Intel Threading Building Blocks [51]. Mainstream languages have also adopted similar features as standard libraries, including Java [36], C++ [53], and Rust [24]. Event-driven programming languages like JavaScript [58] and Dart [1], while they are semantically sequential languages, still adopt a asynchronous task-based model to schedule callbacks on an event loop, illustrating that the task view of computation is useful as a way of thinking about code in contexts beyond concurrent programming.

Languages vary in how the result of one task may be communicated to the others, generally trading off between flexibility of synchronization and ease of writing correct code. Highly structured mechanisms, such as spawn-sync [31] and async-finish [17], are deadlock-free by construction but have limited expressivity. A more flexible mechanism is the *future* [43, 17, 15]. When an asynchronous task is issued, a future object is returned to the parent task immediately. This object is a handle on the asynchronous task and has a *get* method which blocks until the task is completed and retrieves the task's return value. Using a future gives the illusion of having made a synchronous function call since the handle can be passed into other computation units

as if it were the value, but the blocking dependence only occurs when the payload value itself is required for a concrete computation. This facility leads to readable code that is not over-synchronized.

A more powerful generalization of the future is the *promise* [5]. Instead of retrieving a task's return value, it retrieves whatever value it is explicitly *set* to. Thus, a task may produce multiple intermediate values by setting multiple promises. Moreover, there is no restriction on which task sets the promise, so the responsibility to supply a payload is not bound to one task, as it is with futures. For additional flexibility, a promise can also be factored into two separate objects, a producer side and a consumer side.

In contrast to these simple primitives, there are also more complex parallel constructs. In particular, we will examine the *phaser* [74], which subsumes point-to-point synchronization such as futures and promises, but also producer-consumer synchronization such as channels and streams, and repeated multi-party synchronization such as cyclic barriers. Tasks may be registered to a phaser in signal-only, wait-only, or signal-wait modes, which control what interactions are required between the task and the phaser. Phase counters are used to identify which round each signal and wait operation refers to.

1.2 Deadlocks as Bugs

Futures and promises may be passed as values through a program so that any task, not just the parent, can await a future or a promise, and any task can fulfill a promise. This power breaks the safe abstraction that task-parallel programming is supposed to provide over concurrent programming and introduces the possibility of concurrency bugs.

It is possible to construct a deadlocked cycle of gets among futures by appealing to some other synchronization mechanism or a data race [22]. Moreover, cycles of

promises and cycles of phasers can be deadlocked directly without a data race or any other mechanism. Such *logical* deadlocks are almost always symptomatic of a software bug.

Logical deadlocks are distinct from resource deadlocks. Resource deadlocks arise when concurrent processes incrementally acquire resources and enter a scenario where the next request by each process cannot be satisfied because the limited resources are already held by the other processes [52, 18]. Such deadlocks may be avoidable by manipulating the schedule to ensure that at least one process can always acquire its resources [18]. In some cases resource deadlocks can also be corrected after they occur by rolling back one or more processes to a checkpoint [32].

Unfortunately, logical deadlocks are more severe as they indicate a circular dependence in the flow of information in a program. If a logical deadlock occurs, and even if a logical deadlock might occur nondeterministically, the program itself is considered to have a bug, since the circular dependence renders that run semantically meaningless. The best that a runtime can do is to raise an exception or abort the program.

Since logical deadlocks are symptomatic of bugs, we can apply the same programming language design principles that are used to mitigate other kinds of bugs. Take two examples from Java. 1) Array bounds checking is a runtime assertion that ensures index-out-of-bounds bugs are identified as such when they occur, rather than silently causing unsafe behavior [38]; with care, the checks can be made inexpensive [10]. Likewise, it is preferable to detect logical deadlocks when they occur, not later, and to do so without introducing runtime overhead. 2) The iterator-based *for* loop is a language feature that automatically constructs the appropriate loop header for the data structure to iterate over, thereby reducing the code surface for out-of-bounds bugs [38]. Likewise, higher levels of abstraction in parallel languages can remove opportunities for writing deadlocks by automatically constructing safe synchronization

patterns.

Both runtime checks and carefully designed linguistic constructs can assist in solving the deadlock problem. It is desirable to write programs using parallel constructs that preclude deadlocks, if not by construction, at least by the application of a simple usage policy that is intuitive, efficiently verifiable by the runtime, not overly restrictive, and assigns specific blame when violations occur.

1.3 Prior Work in Deadlock Freedom

An ideal parallel programming environment is one in which logical deadlocks are impossible and yet complex computations can be easily expressed without over-synchronization. Deadlock-free settings exist, but are limited in the kinds of parallel control flow that can be expressed [31, 17], impose high burdens on the programmer to supply additional information through annotations or types [12, 54], or require undecidable compile-time analyses [64].

A very inexpensive and non-invasive approach to deadlock detection irrespective of the parallel constructs used is to watch for quiescence of all tasks. This technique is used in the Go language [64], for example. However, deadlocks cannot be found unless and until every remaining task participates in a deadlock, which is not acceptable for long-running programs.

It is, of course, possible to directly construct a graph of blocking dependences and search for the creation of a cycle every time an edge is added. Dynamically guaranteeing deadlock freedom for unrestricted barrier and phaser synchronization through precise cycle detection is a high-cost check and the target of the Armus tool [19]. It is a non-trivial task to correctly maintain this graph under concurrent modifications, which can lead to expensive overheads that are unnecessary in specific settings where more is known about the dependences [22]. Raising an alarm when a cycle forms is suboptimal for error resolution and debugging since the dependence

which arrives last, thereby completing the cycle, is arbitrary and not necessarily the faulty dependence. When the objective is to design a parallel programming environment in which writing deadlock-free code is natural and finding blame for any deadlocks that do arise is easy, precise cycle detection may not be the most effective solution.

The *spawn-sync* mechanism of Cilk [31] is an example of a setting that is deadlock-free because of its limited parallel expressivity. Tasks may only await the completion of all transitively spawned subtasks as a group, and moreover, this synchronization forcibly occurs at the end of every function call, whether requested or not. Deadlock freedom follows naturally by induction on the tree structure of function calls. A more flexible successor to this mechanism is *async-finish*, found in X10 [17] and Habanero-Java [15], where the tree structure is under the control of the programmer through nested *finish* blocks.

It is known that futures can be deadlocked through the use of a data race or another synchronization mechanism [22]. Cogumbreiro et al. developed Known Joins, a deadlock-freedom policy for futures that has a low-overhead runtime implementation [22]. The policy checks each blocking wait operation, permitting only those awaits on futures which are visible to a task according to a set of simple rules that follow from the structure of the program itself.

Unlike futures, which have been retroactively subjected to deadlock freedom [22], phasers came equipped with a deadlock-freedom policy at their first introduction [74]. This policy consists of some interface-level restrictions and some simple checks that occur only when a new task is spawned, rather than at every blocking wait. For example, tasks must not wait on individual phasers, since waiting on them in the wrong order with respect to the signals that are owed can easily construct a deadlock. Instead, a coarse-grained *next* operation automatically signals any outstanding phasers and then awaits the subsequent phase of each phaser.

1.4 Summary of Contributions

Thesis. *The set of task-parallel programs and synchronization mechanisms whose deadlock freedom is guaranteed or verifiable can be expanded through the addition of low-overhead runtime support, novel syntactic constructs, and extended synchronization semantics.*

We develop language features that lead to safe, understandable, and hygienic parallel code, by designing parallel constructs and enforcing principled usage policies that

1. are provably deadlock-free,
2. admit large classes of deadlock-free synchronization,
3. can be checked dynamically with low execution time and memory overheads,
4. and cause complex synchronization patterns to be expressed in a modular, block-structured manner.

This work applies the preceding principles to achieve deadlock-free task-based parallelism in the contexts of futures, promises, and phasers.

- *Transitive Joins* generalizes prior work on deadlock-free policies for futures by admitting an additional class of deadlock-free programs and reducing the amount of dynamic bookkeeping, as its verification algorithm is stateless with respect to blocking waits [83].
- We provide deadlock-freedom verification for promises by introducing annotations for asynchronous tasks to track *ownership* of promises by tasks. We define a policy for ensuring that every promise is fulfilled by its owning task and introduce an algorithm for cycle avoidance among promises that relies on the ownership relation [84].

- To obtain better deadlock avoidance complexity bounds for promises, we define an *approximate* policy that eliminates all deadlocks by appealing to a task ordering. We further introduce a novel pseudo-synchronization primitive, the *guard* block. Guards allow the programmer to demonstrate the safety of low-level code that is difficult to verify by introducing coarse-grained, block-structured synchronization dependences.

- We define the syntax and semantics of *subphase* blocks, which relaxes the already deadlock-free phaser construct by allowing grouped synchronization patterns that exhibit better isolation and modularity, through the use of a tiered phaser organization. Introducing subphase blocks improves parallelism and performance without compromising deadlock freedom.

We address these three primitives (futures, promises, and phasers) because they represent a range of sophistication and expressivity in task-parallel synchronization and are foundational for structurally deterministic computations. Other related communication mechanisms, such as async-finish, channels, or cyclic barriers, are implementable using futures and promises or are special cases of phasers. Futures are the most general of the task-termination primitives because they subsume the family of computation graphs attainable by spawn-sync and async-finish [31, 17]. A future may be thought of as a special case of a promise, since promises allow the payload to be supplied at an arbitrary program point. This grants the versatility to, in principle, construct a phaser out of many promises. (See Listing 3.4, a repeated-use producer-consumer channel, for a start.) Coming from the opposite direction, phasers are, in contrast, complex enough to subsume the other, simpler primitives. Futures and promises can both be rendered as special cases of phasers: A future is essentially a phaser with only one signaling task that signals the phaser upon termination. A promise is essentially a phaser with only one signaling task that signals the phaser once at an arbitrary time.

Why study separate deadlock solutions for all three of these primitives if they are related to each other in these ways? Futures enjoy additional structure that promises do not. Namely, the fulfillment of a future is always tied to a given task's termination. As we will see, avenues for addressing deadlocks among futures can take advantage of this structure to yield a more satisfactory solution that is tailored to futures than if we treated futures as mere promises. Conversely, any solutions that take full advantage of the additional structure of futures are necessarily insufficient when applied to promises, where this structure is absent. To solve deadlocks for promises, we will need to introduce some novel structure. The difficulty with phasers is entirely different still. Phasers have so much structure that they can be made effectively deadlock-free by construction—at a cost to expressivity, efficiency, and modularity. If we strip away all this structure by reducing phasers to a promise-based implementation, we lose the natural deadlock-freedom guarantee. The novel deadlock solution in this work will instead refine the existing structure of phasers to improve modularity and parallelism while retaining a deadlock-freedom guarantee.

CHAPTER 2

DEADLOCK FREEDOM VERIFICATION FOR FUTURES

2.1 Introduction

A logical deadlock bug arises in a parallel program when a cycle of tasks forms in which each task will provide data to the next task but only after receiving data from the preceding task. A simple setting in which deadlocks can arise is the use of *futures* for asynchronous computation results, especially when the futures themselves are shared through data races or other synchronization mechanisms [22]. Deadlocks are difficult to reason about because they can involve arbitrarily many concurrently executing tasks, one or more of which may be to blame, and can arise nondeterministically in racy programs due to variations in task scheduling. Precise cycle detection is not always a desirable solution to the deadlock problem because it either assigns blame to whichever task arrives last, which is arbitrary and nondeterministic, or to the entire cycle, which can be an unhelpfully large amount of information. As an alternative, we present a run-time verifiable policy for futures that rejects specific blocking gets as invalid if they do not conform to a simple set of intuitive organizational rules.

The hierarchical structure of the task spawn tree is a natural source of organizational information that can be used to guarantee deadlock freedom. Indeed, the spawn-sync mechanism of the Cilk language [31] and the async-finish blocks of X10 [17] and Habanero-Java [15], enforce the invariant that every waits-for dependence is a task waiting on the termination of one of its descendants in this tree. Thus, the acyclicity of the waits-for graph and, hence, deadlock freedom follow for free from

Listing 2.1: Single-task deadlock of futures, via a data race.

```
1 Future<Void> [] f = new Future<Void> [1];
2   f[0] = async {
3   f[0].get();
4 };
```

the task tree.

A future, which is a handle representing the eventual return value of a task, can be passed around a program as a value itself. Unlike async-finish, the expressivity of futures allows one to escape the structure of the task tree and create cycles. The basic mechanism for deadlocking futures is given in Listing 2.1. Line 1 allocates a shared reference to a future, which is populated with the handle created on line 2. A write-read race occurs between line 2 and line 3. Provided that the write becomes visible to the read, the asynchronous task will then block, awaiting its own termination and creating a deadlock with itself. In general, deadlocked futures arise through out-of-band sharing of futures. That is, to create a cycle, a future must be passed to a task through a mechanism *other than* through another future [22]. Such mechanisms include data races, properly synchronized nondeterminism, and promises.

The edges of a deadlock cycle alternately consist of waits-for edges (from a blocked task to the resource it awaits) and happens-after edges (backward from the point the resource is supplied to another blocking wait). At run-time, the waits-for edges become explicit, but the happens-after edges do not because they involve code points that have not been, and indeed will not ever be, reached. In some settings, each resource is known to have one owning task, which is responsible for supplying the resource. This is the case when considering futures, since one and only task will supply the future's value upon termination. Therefore, we recover the happens-after edges implicitly and can think of logical deadlocks as merely dependence cycles among tasks.

The deadlock problem for futures is applicable to a wide range of parallel programming models. For example, the forking and joining of system-level threads, as in Java and C++, can be modeled abstractly with tasks futures. Moreover, the concurrency library of Java [36] and, more recently, the standard library of C++ [53] directly include futures as a high-level abstraction. Cilk's spawn-sync model is more limited than futures because a Cilk function is compelled to join with all the tasks it has spawned [31]. Cilk programs can only exhibit *fully strict* computation graphs [9]. X10 [17], the Habanero Java language (HJ) [15], and the Habanero Java Library (HJlib) [50] include an async-finish model, which, although more general than Cilk, is also more limited than futures; rather than join with arbitrary tasks, a task can join all at once with the collection of tasks created transitively within a given computation. Programs based on async-finish exhibit *terminally strict* computation graphs [39]. X10, HJ, and HJlib also support arbitrary joins with asynchronously spawned tasks in the form of futures [17, 15, 50].

Approaches to the deadlock problem include solutions which statically detect the possibility of deadlocks or verify deadlock freedom [87, 60, 12], solutions which detect deadlocked tasks at runtime [56, 47], and solutions which avoid deadlocks by intercepting blocking operations that might cause a deadlock if allowed to proceed [62, 19, 22]. Our work can be used as a deadlock-avoidance policy that gives target programs the ability to handle illegal joins as a runtime exception.

2.1.1 Contributions

Following the tradition of using the task tree structure to guarantee deadlock freedom via fully strict or terminally strict computation graphs, we develop a novel deadlock-freedom policy with an efficient verifier that imposes fewer restrictions than prior work. Specifically, this paper makes the following contributions:

1. We formulate a policy called *Transitive Joins* (TJ) as a simple set of rules based

on the task tree that restricts which joins (blocking get operations) are permissible, and we prove that TJ guarantees a program's joins will not deadlock.

2. We prove that TJ extends an existing policy due to Cogumbreiro et al. [22], called Known Joins (KJ), by admitting an additional class of programs that are not valid under KJ. We argue for the utility of the large class of the newly admitted TJ programs.

3. TJ can be used as a runtime verification algorithm that aborts potentially unsafe joins by raising an exception. A simple algorithm for TJ verification takes $\mathcal{O}(h)$ time to check each join and $\mathcal{O}(n)$ space in total, where n is the number of tasks spawned, and h is the height of the task tree.

4. We evaluate a TJ verifier implementation, showing a geometric mean overhead of 1.06× execution time and 1.09× memory usage, justifying the use of TJ in practice as an always-on runtime safety check. TJ's overheads improve on an implementation of KJ, whose time and memory overheads are 1.09× and 1.30×, respectively.

Our Transitive Joins verifier implements one of three proposed algorithms for the policy. We evaluate the TJ verifier against two available KJ verifiers on five benchmarks which both TJ and KJ admit as valid. We also include a deadlock-free benchmark which satisfies TJ but not KJ, thereby requiring the KJ verifiers to fall back to a more expensive cycle-detection algorithm. The results show that TJ incurs comparable or better execution time and memory overheads to the KJ implementations, so that it is practical and desirable to use TJ rather than the KJ verifier and classical cycle-detection.

2.1.2 Outline

Section 2.2 gives an informal description of TJ and explores its utility through example programs. In Section 2.3 we give formal definitions and theorems on the correctness of TJ, and we formally relate TJ to KJ and async-finish. Section 2.4 describes a TJ verification algorithm, which is evaluated in Section 2.5. Section 2.6 covers related work on deadlocks, and Section 2.7 concludes.

2.2 Overview

We informally describe our novel Transitive Joins policy and frame it as the transitive closure of two intuitive rules. Through two example programs, we illustrate the utility of TJ in admitting an additional class of deadlock-free programs over prior work.

2.2.1 TJ Principles

Because precise cycle detection is inadequate, we accept the reality of false positives in deadlock freedom strategies. So in designing a safe policy (one that rejects at least all deadlocking runs) it is important to carefully control *which* runs raise false positives. The join operations that are permitted by a safe policy ought to be those which naturally emerge from program structure, for two reasons. 1) It is inconvenient for the programmer if the policy excludes a program whose deadlock freedom is plainly evident. 2) By excluding programs whose deadlock freedom is obscure and hard to understand, we encourage sanitary coding practices.

Guided by program structure, we devise a few principles which determine the join permissions that a given task is granted. First, we observe that in the async-future model of task parallelism, the continuation of a parent task receives a future which refers to the forked task; that is, the parent has a joinable handle to the child. However, the child task does not receive a handle to the parent. Therefore, it is

natural to

1. permit the parent to join with (wait for) the child, but not to permit the child to join with the parent.

Second, a forked task receives parameters or captured data from the parent task, which can readily include previously created futures. Therefore, it is natural to

2. permit a child to join with the tasks for which the parent held join permission at the time of the fork.

Prior work takes rules 1 and 2 and adds a third rule to create a safe deadlock-freedom policy called Known Joins (KJ) [22]. A corollary to the safety of KJ is that rules 1 and 2 alone are also safe (though very constraining).

The key observation which leads to our novel policy is the following: Under a safe policy, if task a performs a permitted join with task b, and if task b performs a permitted join with task c, then task a has effectively blocked on task c and yet is not in danger of deadlocking. Therefore, it is natural to declare that

3. permission-to-join should be a transitive relation.

The Transitive Joins policy (TJ) may be informally defined as the preceding three rules.

Figure 2.2 illustrates two programs and their TJ permissions. First consider the diagram on the left. Task a forks task b, then task d. Task b forks task c. There is no guarantee about whether c or d is created first. Under rule 1, it is always valid for a parent to join with its own child. Therefore, every fork edge is also a join permission edge. By rule 2, task d inherits from a its permission to join on b, since a held this permission at the time d was created. Readers familiar with KJ will see that, after d joins with b, d then *learns* KJ permission to join with c. But under rule 3 of TJ, d has permission to join with c by transitivity through b, whether or not d actually joins with b.

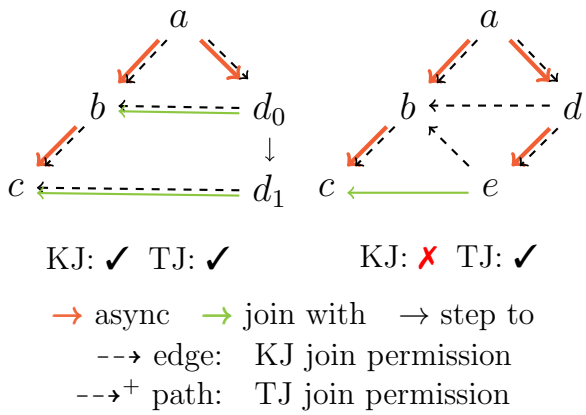

Figure 2.2: The actions of two example programs, showing the task tree (orange), the joins (green), and the join permissions (dotted). Children of each node are drawn in spawn order from left to right. The run on the right is accepted only by TJ.

The diagram on the right of Fig. 2.2 begins with the same scenario of forks. However, d then forks task e, which joins with c. Task e inherits from its parent the permission to join with b. Under KJ it is not legal for e to directly join with c.[1] By contrast, join permission is transitive in TJ. Since there is a (non-empty) path of join permissions from e to c, e is permitted by TJ to join with e *without* first joining on the other nodes along that path (namely b).

Compared to the state of the art, Transitive Joins reduces the gap between what is deadlock-free and what is accepted by a policy. We will show that KJ-valid executions are a strict subset of TJ-valid executions (Theorem 20). TJ admits new executions which rely on the transitivity of join permission but which skip joins or perform joins out of order with respect to KJ. Stated altogether as a single principle, TJ allows *any task* in a subtree T_1 of the task tree to join with *any task* in another subtree T_2, provided the root of T_2 is an older sibling of (same parent, but spawned earlier than) the root of T_1. This principle will be formalized as Theorem 15.

[1]KJ was designed to prove deadlock freedom from data race freedom in the absence of additional synchronization mechanisms. If e obtains a handle to c without synchronization, there must be a data race, so KJ is not interested in admitting this execution.

Listing 2.3: Divide-and-conquer algorithm with no guarantee on the relative order of joins.

```
1 int f (Queue<Future> tasks) {
2   if (done()) return 1;
3   // Task launches before future is added,
4   // so its children may appear before or after it.
5   tasks.add(async { return f(tasks); });
6   tasks.add(async { return f(tasks); });
7 }
8 void main () {
9   Queue<Future> tasks =
10      new ConcurrentLinkedQueue<Future>();
11  int result = f(tasks);
12  while (!tasks.isEmpty()) {
13    // May join with any descendant.
14    result += tasks.poll().get();
15  }
16 }
```

2.2.2 Program Model

We use the **async** keyword to asynchronously execute a block of code (copying captured local values, but sharing heap-allocated objects and arrays); **async** immediately returns a handle to the task, called a future. Futures have a blocking method, **get**, which waits for the associated task to terminate (joins with it) and then returns that task's result value (if the type is not void). Our examples make use of some standard Java classes and methods. In particular, ConcurrentLinkedQueue is a queue for which concurrent accesses are sequentially ordered, and AtomicReferenceArray is an array whose entries exhibit volatile access.

2.2.3 Unordered Join with All Descendants

To see the utility of a transitive permission-to-join relation, consider the program given in Listing 2.3. It consists of the skeleton of a divide-and-conquer algorithm, f, which we have parallelized by enclosing each recursive call in its own **async** task on line 5 and line 6. We keep the future for every task in a shared queue, and the root task

awaits the completion of the entire algorithm by joining on each future in the queue (lines 12–14). The join pattern of this program behaves like an implementation of the finish construct, where the root joins with all tasks transitively spawned during its computation.[2] Because a **get** does not unblock until the awaited task terminates, it is guaranteed that once the queue of tasks is found to be empty, no more asynchronous tasks are running.

However, when each new task is created, it may begin executing before or after its future is placed onto the queue. Therefore, the queue does not respect any ordering between parent and child tasks. Many possible runs of this program violate the Known Joins policy because the root task may try to join with a task, a, before obtaining permission to do so via joining on the parent of a. Such a scenario does not, however, violate the Transitive Joins policy since TJ employs a transitive join permission. If the root is permitted to join with the parent of a, since the parent is permitted, in turn, to join with a, then it follows that the root is permitted to join with a directly. Therefore, the root is permitted to join with an arbitrary element of the queue at any time.

2.2.4 Critical Path Reduction

To further show the utility of the permissions granted by Transitive Joins, we present a program implementing a map-reduce algorithm. Map-reduce is a common pattern of concurrency in which a large collection of parallel work is distributed among several mapper tasks, later to be accumulated into one result by one or more reducer tasks. We give an example map-reduce program in Listing 2.4. Lines 5–6 fork N asynchronous mappers, but since these lines of code are themselves asynchronous to the root task, the program does not wait for all the mappers to be created before continuing. Lines 9–20 spawn C asynchronous reducers. Each reducer accumulates the

[2]More specifically, the join pattern is a natural way to implement the 'finish accumulator' construct [73], which joins with all tasks that were forked within some scope and collects their results.

Listing 2.4: Map-reduce program showing another use of Transitive Joins.

```
1 void main () {
2   AtomicReferenceArray<Future> mappers =
3       new AtomicReferenceArray<Future>(N);
4   async { // Async mapper spawning
5     for (i = 0; i < N; i++)
6       mappers.set(i, async { return work(); });
7   };
8   // Chunked reduce phase
9   Future[] reducers = new Future[C];
10  for (c = 0; c < C; c++) {
11      reducers[c] = async {
12          acc = 0;
13          for (i = c*N/C; i < (c+1)*N/C; i++) {
14              while (mappers.get(i) == null)
15                  Thread.yield();
16              acc += mappers.get(i).get();
17          }
18          return acc;
19      };
20  }
21  acc = 0;
22  for (c = 0; c < C; c++)
23      acc += reducers[c].get();
24 }
```

results of a chunk of N/C mappers in lines 12–18.[3] Finally, lines 21–23 accumulate the partial results from all reducers.

Listing 2.4 is deadlock-free, and it is valid under TJ but not under KJ. Observe that the mapper tasks are grandchildren of the root, and the reducers inherit the root's join-permission relationship to the mappers. According to KJ it is illegal to execute the blocking **get** on line 16, unless 1) the root joins with the line 4 task *prior* to spawning the reducers, or else 2) each reducer must itself join with the line 4 task. Unlike Listing 2.3, Listing 2.4 *always* violates KJ, rather than violating it nondeterministically.

However, under TJ it is legal for the reducers to join with the mappers without any additional requirements. The root task is transitively permitted to join with

[3]This pattern is more complex than that of Listing 2.3 and cannot be written using finish constructs because each reducer selects a subset of mappers to wait on, rather than all of the tasks that were forked within a given scope.

its grandchildren (the mappers), and the reducers inherit that permission. In this way, the program can begin reducing the results as soon as they arrive, even if they arrive before all the mapper tasks are spawned. That is, a KJ-compliant variant of Listing 2.4 would have a longer critical path than the present code since a join with the line 4 task would have to be inserted into the critical path.

2.3 Policy Formalism

We formally define our novel Transitive Joins policy and prove three main results:

1. TJ join permission is a total order, so all TJ-valid program runs are deadlock-free (Theorems 10, 11).

2. TJ join permission is induced by the preorder traversal of the task tree and has a natural decision procedure based on lowest common ancestors (Theorems 15, 17).

3. TJ join permission admits every Known Joins program and every async-finish program (Theorems 20, 23).

2.3.1 Policy Definition

The Transitive Joins policy is defined over a program trace language and recognizes as valid a conservative subset of deadlock-free traces.

Definition 1. Let symbols $a, b, \ldots$ denote *tasks*. An *action*, α, is one of $\mathit{init}(a)$ (a is the root task), $\mathit{fork}(a, b)$ (a spawns b), or $\mathit{join}(a, b)$ (a awaits the termination of b). A *trace*, t, is a sequence of actions. We will use $t_1; t_2$ for trace concatenation.

A trace is valid if it satisfies reasonable constraints on the tasks used in the forks and joins. For example, a fork should always introduce a new task, and a join should only occur between certain pairs of tasks, according to the policy. We keep the

formalism general so that other policies (in particular, Known Joins) can fit into the same framework.

Definition 2. Let R be some family of relations indexed by traces so that R_t is the relation corresponding to trace t. With respect to R, let $t : A$ denote that t is a valid trace consisting of the tasks A, as follows.

$$\frac{}{\mathit{init}(a) : \{a\}} \text{ valid-init}$$

$$\frac{t : A \qquad a \in A \qquad b \notin A}{t; \mathit{fork}(a, b) : A \cup \{b\}} \text{ valid-fork}$$

$$\frac{t : A \qquad R_t(a, b)}{t; \mathit{join}(a, b) : A} \text{ valid-join-}R$$

The valid-* rules state that a trace must begin with an *init* action, that each *fork* action must connect an existing task with a fresh task, and that each *join* action must connect two tasks which are related by R_t, where t is the trace so far.

Definition 3. The Transitive Joins judgment $t \vdash a < b$ is given by the following rules. Let $t \vdash a \leq b$ be shorthand for $a = b \vee (t \vdash a < b)$.

$$\frac{t \vdash c \leq a}{t; \mathit{fork}(a, b) \vdash c < b} \text{ TJ-left}$$

$$\frac{t \vdash a < c}{t; \mathit{fork}(a, b) \vdash b < c} \text{ TJ-right}$$

$$\frac{t_1 \vdash a < b}{t_1; t_2 \vdash a < b} \text{ TJ-mono}$$

The $<$ relation should be regarded as the permission-to-join relation for the Transitive Joins policy. (We will show that $<$ is a total order over all the tasks in a trace, justifying the choice of notation.)

Definition 4. The *Transitive Joins policy* (TJ) accepts those traces $t : A$ that are

derivable when the relation family R is instantiated as $R_t(a, b) \triangleq t \vdash a < b$.

2.3.2 TJ is Deadlock-free

Lemma 5 (Irreflexivity)**.** *For any TJ-valid trace $t : A$, $t \vdash a < a$ cannot be derived for any $a \in A$.*

Proof. By induction on t.

a) $t = \mathit{init}(a) : \{a\}$: No TJ-* rules apply.

b) $t = t'; \mathit{fork}(a, b) : A$ and $t' : A'$: Let $a' \in A$ be given, and suppose for the sake of contradiction that we derived $t \vdash a' < a'$ using rule

   i) TJ-left: We require $a' = b$ and $t' \vdash a' \leq a$. Since $b \notin A'$, we have neither $t' \vdash b < a$ nor $b = a' = a$.

   ii) TJ-right: We require $a' = b$ and $t' \vdash a < a'$. Again since $b \notin A'$, we cannot have $t' \vdash a < b$.

   iii) TJ-mono: We require $t' \vdash a' < a'$ and thus $a' \in A'$. But by the inductive hypothesis, $t' \vdash a' < a'$ is not derivable.

c) $t = t'; \mathit{join}(a, b) : A$: Since t ends with a join, only TJ-mono could derive $t \vdash a' < a'$ for some $a' \in A$; this rule requires $t' \vdash a' < a'$, which is not derivable by the inductive hypothesis.

□

Lemma 6. *If $t : A$ (w.r.t. any relation family R) and $a \in A$, there is a unique action $\alpha = \mathit{fork}(p, a)$ occurring in t, for some task p.*

Proof. By induction on $t : A$, observing the invariant that $\mathit{fork}(\cdot, b)$ is the only action that can add b to the set, A, and, moreover, in order to append $\mathit{fork}(\cdot, b)$, b must not already be in A, by the hypothesis of valid-fork. □

Definition 7. In the statement of Lemma 6, let p be called the *parent* of a, and a a *child* of p. Let the (irreflexive) transitive closure of the parent relation be the *ancestor* relation. The reverse of the ancestor relation is the *descendant* relation.

Lemma 8 (Transitivity of $<$)**.** *For each TJ-valid task $t : A$ and $a, b, c \in A$, if $t \vdash a < b$ and $t \vdash b < c$, then $t \vdash a < c$.*

Proof. By induction on t.

a) $t = \mathit{init}(a) : \{a\}$: Vacuous by Lemma 5.

b) $t = t'; \mathit{fork}(a, b) : A$ and $t' : A'$: By the inductive hypothesis and TJ-mono we have that $t \vdash \cdot < \cdot$ is transitive over A'. It remains to include b in the transitivity. Where $x, y \in A'$, we find that we can only derive

- $t \vdash x < b$ from TJ-left and $t' \vdash x \leq a$
- $t \vdash b < x$ from TJ-right and $t' \vdash a < x$
- $t \vdash x < y$ from TJ-mono and $t' \vdash x < y$.

There are three roles b can play in the transitivity property; the other two roles (call them a_1, a_2) are distinct from b (and thus in A') by Lemma 5.

i) Suppose $t \vdash a_1 < a_2 < b$. Then $t' \vdash a_1 < a_2 \leq a$, which yields $t' \vdash a_1 \leq a$ by the existing transitivity. With TJ-left we derive $t \vdash a_1 < b$, as desired.

ii) Suppose $t \vdash b < a_1 < a_2$. Then $t' \vdash a < a_1 < a_2$, which yields $t' \vdash a < a_2$ by the existing transitivity. With TJ-right we derive $t \vdash b < a_2$, as desired.

iii) Suppose $t \vdash a_1 < b < a_2$. Then $t' \vdash a_1 \leq a < a_2$, which yields $t' \vdash a_1 \leq a_2$ by the existing transitivity. With TJ-mono we derive $t \vdash a < a_2$, as desired.

c) $t = t'; \mathit{join}(a, b) : A$: Trivial by the inductive hypothesis and TJ-mono.

□

We now show that TJ is *safe* in the sense that it does not admit any deadlocking traces.

Definition 9. A trace t contains a *deadlock* if there is a sequence of tasks $a_0, a_1, \ldots, a_n$ ($n \geq 0$) such that t contains $\mathit{join}(a_n, a_0)$ and $\mathit{join}(a_i, a_{i+1})$ for all $i < n$.

Theorem 10 (Total order)**.** *For any TJ trace $t : A$, the relation $t \vdash \cdot < \cdot$ defines a (strict) total order over A. That is, $<$ is transitive and trichotomous (for all $a, b \in A$, exactly one of $a < b$, $a = b$, $b < a$ holds).*

Proof. We already have transitivity (Lemma 8). It remains to show trichotomy. First, $a = b$ precludes $t \vdash a < b$ and $t \vdash b < a$ by Lemma 5. Second, if $a \neq b$, we cannot have both $t \vdash a < b$ and $t \vdash b < a$, since transitivity would then yield $t \vdash a < a$. It remains to show that if $a \neq b$, then at least one of $t \vdash a < b$ or $t \vdash b < a$ holds. We prove the property by induction on t.

a) $t = \mathit{init}(a) : \{a\}$: Vacuous.

b) $t = t'; \mathit{fork}(a, b) : A$: Since only b is new, it suffices to show that for all $a' \in A$ $a' \neq b$ implies $t \vdash a' < b$ or $t \vdash b < a'$. By the inductive hypothesis, we have $t' \vdash a' \leq a$ or $t' \vdash a < a'$. In the first case, apply TJ-left; in the second, apply TJ-right.

c) $t = t'; \mathit{join}(a, b) : A$: Trivial by the inductive hypothesis and TJ-mono.

□

Theorem 11 (Deadlock-freedom of TJ)**.** *If t is a TJ trace, then t does not contain a deadlock.*

Proof. To derive the TJ validity of t, each $\mathit{join}(a, b)$ requires the rule valid-join-R and the hypothesis $R_{t'}(a, b)$, that is, $t' \vdash a < b$, where t' is the trace so far. By TJ-mono,

we can then derive $t \vdash a < b$ by completing the trace. Since the $<$ relation is a total order (Theorem 10), we do not have a deadlock cycle. □

2.3.3 TJ Order as a Tree Traversal

The preceding formalism has shown that TJ guarantees deadlock-freedom; however, it is not immediately clear how to implement the policy. We now characterize TJ in a way that suggests a natural decision procedure, and we prove its equivalence to the original definition.

Definition 12. Let a trace $t : A$ be given, and let $E = \{(a, b) \mid \mathit{fork}(a, b) \in t\}$ serve as the (directed) edge relation of a tree, T, over the vertices A. Further, let T be equipped with a local child indexing function, $I : A \to \mathbb{Z}$, behaving as follows. For children, b_1 and b_2 of a, $\mathit{fork}(a, b_1)$ precedes $\mathit{fork}(a, b_2)$ in t if and only if $I(b_1) < I(b_2)$. T is called the *task tree* of t.

In the remainder of the section, let T be the task tree of a TJ-valid trace $t : A$.

Definition 13. Define a *preorder tree-traversal* $<_T$ to be any total order over A satisfying the following rules:

1. If $\mathit{fork}(a, b)$ is in t, then $a <_T b$.
2. If $\mathit{fork}(a, c)$ precedes $\mathit{fork}(a, b')$ in t, and b' is an ancestor of b, then $b <_T c$.

Definition 14. With respect to a tree, let $\mathit{lca}^+(a, b)$ denote the *extended lowest common ancestor* of a and b, defined as

1. $\textsc{anc}^+$ when a is a (proper) ancestor of b,
2. $\textsc{dec}^*$ when a is a descendant of or equal to b, or
3. $\textsc{sib}(a', b')$, where a', b' are the unique nodes such that a', b' are siblings, a' is an ancestor of a, and b' an ancestor of b.

Note that the traditional lowest common ancestor of a and b is either a, b, or the parent of a' and b', corresponding to each of the three cases of lca^+. The extended version has the benefit of telling us how each of a and b are connected to their lowest common ancestor.

Theorem 15 (Decision procedure for $<_T$)**.** *Let $a, b \in A$ be given. Proceeding case-wise on $lca^+(a, b)$, we have*

a) ANC^+ *implies* $a <_T b$.

b) DEC^* *implies* $a \not<_T b$.

c) $\text{SIB}(a', b')$ *implies* $(I(a') > I(b') \iff a <_T b)$.

Proof. The cases are

a) ANC^+: Induct on the path from a to b, using rule 1 of Definition 13 and transitivity to get $a <_T b$.

b) DEC^*: Either $a = b$, or by reversing the roles we can use ANC^+ to show $b <_T a$. Both scenarios demonstrate $a \not<_T b$ by trichotomy.

c) $\text{SIB}(a', b')$ and $I(a') > I(b')$: Since b' is an ancestor of b, we have already seen that $b' \leq_T b$. Apply rule 2 of Definition 13 to obtain $a <_T b' \leq_T b$, and then invoke transitivity.

d) $\text{SIB}(a', b')$ and $I(a') < I(b')$: Reverse the roles of a and b in the previous case, and then apply trichotomy.

□

Corollary 16. *There is at most one $<_T$.*

Theorem 17 (Preorder)**.** $t \vdash \cdot < \cdot$ *is the unique $<_T$.*

Proof. $t \vdash \cdot < \cdot$ satisfies Definition 13 rule 1 by TJ-left (and TJ-mono). $t \vdash \cdot < \cdot$ satisfies Definition 13 rule 2 by induction on the (non-empty) path from a to b' to b by applying TJ-right at each fork on the path (and TJ-mono as necessary). Therefore, $t \vdash \cdot < \cdot$ is a $<_T$ relation, and, by Corollary 16, the unique such relation. □

Therefore, Theorem 15 suggests an algorithm for TJ verification based on lowest common ancestors in the task tree. One can dynamically construct T and I as a monotonically growing data structure during the execution of a trace. Upon executing $\mathit{fork}(a, b)$, b is added to T as a new child of a. The index map, I, represents the order in which the children are added.

2.3.4 TJ Subsumes KJ and async-finish

We can represent the definition of the Known Joins policy [22] within the framework of Section 2.3.1. The common setting then allows us to prove that Transitive Joins subsumes Known Joins (Theorem 20), meaning that TJ accepts at least all of the same executions as KJ.

Definition 18. The judgment $t \vdash a \prec b$, which denotes that task a *knows* task b after the execution of trace t, is given by the following rules.

$$\frac{}{t; \mathit{fork}(a, b) \vdash a \prec b} \text{ KJ-child}$$

$$\frac{t \vdash a \prec c}{t; \mathit{fork}(a, b) \vdash b \prec c} \text{ KJ-inherit}$$

$$\frac{t \vdash b \prec c}{t; \mathit{join}(a, b) \vdash a \prec c} \text{ KJ-learn}$$

$$\frac{t_1 \vdash a \prec b}{t_1; t_2 \vdash a \prec b} \text{ KJ-mono}$$

Definition 19. The *Known Joins policy* (KJ) instantiates the relation family R as $R_t(a, b) \triangleq t \vdash a \prec b$, and accepts each trace t for which we can derive some $t : A$.

The $\prec$ relation is the permission-to-join, or "knowledge," relation of Known Joins. The above definition is modified from its original form in Cogumbreiro et al. [22]. The original definition used a map K from tasks to knowledge sets. We have $a \prec b \iff b \in K(a)$. The rules KJ-child and KJ-inherit show, respectively, that a parent task knows its child, and the child inherits all the knowledge of the parent at the time of the fork (cf. rule T-ASYNC, [22]). The rule KJ-learn shows that, in a join, the waiting task acquires all the knowledge of the terminating task (cf. rule T-GET, [22]). From KJ-mono, we have that $\prec$ grows monotonically during a trace.

Note the parallels between the $<$ rules in Section 2.3.1 and the $\prec$ rules here. In particular, TJ-left subsumes KJ-child; TJ-right is essentially KJ-inherit; and we have monotonicity in both cases by the *-mono rules. The differences are that TJ-left obtains much more information than KJ-child (completing transitivity in combination with TJ-right) and that $<$ has no join rule.

We say that TJ *subsumes* KJ in the following result.

Theorem 20. *If t is KJ-valid, then $t \vdash a \prec b \implies t \vdash a < b$.*

Proof. By induction on the proofs of $t \vdash a \prec b$ and of the KJ-validity of t. If $t \vdash a \prec b$ is derived by KJ-child, KJ-inherit, or KJ-mono, replace the given rule with TJ-left, TJ-right, or TJ-mono, respectively, and recurse on the hypothesis of the rule. The remaining case is that $t \vdash a \prec b$ is derived by KJ-learn. We, therefore, must have a proof of that rule's hypothesis, $t' \vdash c \prec b$, where $t = t'; \mathit{join}(a, c)$. Recurse on $t' \vdash c \prec b$ to obtain $t' \vdash c < b$. Since t is KJ-valid and ends with a join, the KJ validity of t is derived by valid-join-R (for $R = \prec$), which has, as a hypothesis, $t' \vdash a \prec c$. Recurse on $t' \vdash a \prec c$ to obtain $t' \vdash a < c$. Finally, apply Lemma 8 (transitivity of $<$) to obtain $t' \vdash a < b$, which yields $t \vdash a < b$ by TJ-mono. □

Corollary 21. *If t is KJ-valid, then t is TJ-valid. That is, the TJ policy accepts a superset of the traces of the KJ policy.*

Note that the superset relation of Corollary 21 is strict, since we have given an example of a program that is TJ-valid but not KJ-valid (Section 2.2.3).

We have now seen that TJ subsumes KJ and that TJ is a total order over the created tasks. TJ is in this sense a maximally permissive safe policy: TJ permits a superset of the KJ-valid traces, but if even a single additional task pair were added to TJ's total order join permission relation, TJ would then admit some deadlocking traces. Therefore any safe policy which strictly subsumes TJ must necessarily not impose a total order *a priori* from the task tree; that is, it must respond dynamically to other events besides task forking. By observing only forks, TJ improves upon KJ (which observes both forks and joins), but to improve upon TJ one must again observe more than forks.

Finally, we can relate TJ to the async-finish construct, which produces terminally strict computation graphs. A finish block is defined to await the termination of every task that is spawned within its scope, directly or transitively.

Definition 22. A trace $t : A$ is an *async-finish trace* if it is derivable when $R_t(a, b)$ is defined to hold when a is a (proper) ancestor of b in T, the task tree of t.

Theorem 23. *If t is an async-finish trace, then t is KJ-valid.*

Proof. If a is an ancestor of b, then $a <_T b$; that is, a precedes b in the preorder traversal of T. Hence $t \vdash a < b$ by Theorem 17. □

2.4 TJ Verifier

We describe an online verifier of TJ validity, including fork and join routines and some possible algorithms for the $<_T$ decision procedure of Theorem 15. There is a modular separation between the fork and join routines of the verifier and the underlying implementation of Theorem 15.

Algorithm 1 Verifier Interface

```
1:  procedure FORK(a, f)
2:      if a = null then
3:          v ← ADDCHILD(null)
4:      else
5:          v ← ADDCHILD(a.node)
6:      b ← {node : v, result : null}
7:      do asynchronously
8:          b.result ← f()
9:      return b

10: procedure JOIN(a, b)
11:     assert LESS(a.node, b.node)
12:     wait(b)
13:     return b.result
```

2.4.1 Verifier Interface

A Transitive Joins verifier can be inserted into a task-parallel runtime by performing a small amount of bookkeeping at each fork and by checking the policy before each join. We supply two procedures in Algorithm 1:

- FORK(a, f) implements the action of **async** f by task a; it returns the new child task. If a is *null*, this represents the initialization of a root task to execute f.
- JOIN(a, b) implements the action of **get** b by task a; it returns the return value of b upon termination of b if the join is TJ-valid, and faults (without blocking) otherwise.

Algorithm 1 makes calls to two procedures which maintain a tree data structure, T:

- ADDCHILD(u) should create and return a new child of vertex u in T, or a new root vertex if u is null.

- LESS(v_1, v_2) should return whether $v_1 <_T v_2$ in the tree T that has been constructed by all the preceding calls to ADDCHILD.

The interface to the verifier is subject to four constraints. The requirements imposed on ADDCHILD and LESS are

1. every call to ADDCHILD must return a unique value, and
2. LESS and ADDCHILD may be called concurrently (with themselves and with each other).

The guarantees provided to ADDCHILD and LESS by Algorithm 1 are

3. no two instances of ADDCHILD shall be called concurrently on the same parameter (since that parameter uniquely determines the calling task), and
4. each parameter to LESS shall previously have been returned by ADDCHILD.

2.4.2 Possible LCA Algorithms

According to Theorem 15, the join validation step in a runtime verifier is essentially a lowest common ancestors computation in the task tree, T. Recall that we defined an extended LCA function, $lca^+(a, b)$, which provides specific information about how a and b are connected to their LCA. We describe three possible algorithms, TJ-GT, TJ-JP, and TJ-SP. The complexity bounds for each algorithm, and for prior work, are recorded in Table 2.1.

TJ-GT The most basic algorithm for maintaining the task tree, T, and answering lca^+ queries is given in Algorithm 2. This algorithm, TJ-GT, is characterized by having a shared global tree. Each vertex, v, stores a pointer to its parent, u, an index indicating how many siblings have preceded v, the depth of v in T, and the number of children v has forked so far.

Algorithm 2 TJ-GT implementation

```
 1: procedure AddChild(u)
 2:     v ← {parent : u, ix : null, depth : 0, children : 0}
 3:     if u = null then
 4:         return v
 5:     v.depth ← u.depth + 1
 6:     v.ix ← u.children
 7:     u.children ← u.children + 1
 8:     return v

 9: procedure Less(v1, v2)
10:     if v1 = v2 then
11:         return false
12:     else if v1.depth < v2.depth then
13:         return ¬Less(v2, v1)
14:     i1, i2 ← null                          ▷ child indices we arrive by
15:     while v2.depth < v1.depth do
16:         i2 ← v2.ix
17:         v2 ← v2.parent
18:     while v1 ≠ v2 do
19:         i1 ← v1.ix
20:         i2 ← v2.ix
21:         v1 ← v1.parent
22:         v2 ← v2.parent
23:     if i1 = null then                      ▷ i2 is never null
24:         return true                        ▷ ANC+ case
25:     return i1 > i2                         ▷ v1 <T v2 iff i1 > i2
```

Per its specification, AddChild(u) creates and installs a new child for vertex u. The first step is to allocate a unique vertex, setting u as its parent (line 2). In the special case that we are creating the root node (u is *null*), we leave v with an initial depth of 0 and no children (lines 3–4). Otherwise, the depth of v is calculated from its parent's depth (line 5). The number of preceding children, ix, is taken from the parent's child count, which is then incremented (lines 6–7).

Less(v_1, v_2) decides $v_1 <_T v_2$. The procedure first eliminates the case that v_1 is at least as deep as v_2 using the equivalence $v_1 <_T v_2 \iff v_1 \neq v_2 \wedge v_2 \not<_T v_1$ (lines 10–13). The procedure then lifts v_2 to an ancestor of the same depth as v_1

Table 2.1: Algorithmic complexities of competing policies for deadlock freedom for futures; n is the number of tasks, and h is the height of the task tree. In the worst case, $h = n$.

	KJ-VC	KJ-SS	TJ-GT	TJ-JP	TJ-SP
Fork time	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(\log h)$	$\mathcal{O}(h)$
Join time	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(h)$	$\mathcal{O}(\log h)$	$\mathcal{O}(h)$
Space	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n \log h)$	$\mathcal{O}(nh)$

by following the parent pointers (lines 15–17). Afterward, both v_1 and v_2 are lifted together until their LCA is found (lines 18–22). Throughout the process, every time a parent pointer is traversed from v to v', the index of v as a child of v' is stored (lines 14, 16, 19–20). If the path traversed by v_1 to the LCA was trivial, since the path traversed by v_2 always takes at least one step, the initial v_1 was a proper ancestor of the initial v_2 (lines 23–24). Finally, i_1 and i_2 give us the relative order of the two sibling vertices that were traversed immediately prior to reaching the LCA (line 25).

AddChild satisfies constraint 1 of Section 2.4.1 because it returns a freshly allocated vertex, v. Observe that no concurrent reads can be performed on the fields of v until after AddChild returns, thanks to constraint 4 of Section 2.4.1. Moreover, the only data that is modified by AddChild and is at the same time visible to other tasks is u.children. However, Less does not access children, and constraint 3 ensures that no concurrent AddChild will access the same u.children. Furthermore, Less is a read-only procedure. Therefore, we justify the correctness of AddChild and Less in satisfying requirement 2 *without* making use of any synchronization on the data structure.

Each vertex (hence each task) requires a constant amount of storage. The space complexity of TJ-GT is then $\mathcal{O}(|T|)$, linear in the number of tasks created. In order to update the data structure upon a fork, $\mathcal{O}(1)$ operations are required. No update is required upon joining, but the join verification may require scanning the tree along

two paths no longer than the height of the tree. Therefore the time complexity per join is $\mathcal{O}(\mathit{height}(T))$.

TJ-JP Alternative algorithms for lowest common ancestors, and hence lca^+, can be formulated based on various solutions to the level-ancestor problem [45]. For the sake of exploring different time and space complexity trade-offs, we consider another possible algorithm, TJ-JP, based on *jump pointers* [7]. At a cost of extra pointers in the tree, we can dramatically improve the join verification complexity. Let each node maintain not a single parent pointer but an array of pointers to each of its 2^ith ancestors. Moreover, let each of these pointers be paired with the index of the child that the pointer arrives through; these will serve the same purpose as the ix field in Algorithm 2. The space per node is no longer constant but $\mathcal{O}(\log \mathit{height}(T))$, since a node at depth d will need an array of size $\log_2 d$. Setting up these jump pointers requires $\mathcal{O}(\log d)$ time per fork at a node of depth d. However, the jump pointers make traversing the tree much more efficient than in TJ-GT. Instead of scanning linearly across two paths to find their meeting point, one can perform a binary search using the jump pointers. Thus, the join verification time is only $\mathcal{O}(\log \mathit{height}(T))$.

TJ-SP Finally, we propose a task-local version of the $<_T$ decision algorithm, TJ-SP (Algorithm 3), in which an explicit shared tree is replaced by a per-task array recording the task's path from the root (its spawn path). Upon each fork, the new task copies its parent's array, appending its own index among its siblings (line 4). An $\mathcal{O}(\log \mathit{height}(T))$ scan for the longest common prefix of two tasks' spawn paths yields lca^+ information (lines 9–12). Where the paths diverge, we compare the sibling indices as usual (lines 10–11). If the paths match up to the length of the shorter one, then one of the tasks is an ancestor of the other, so we use the relative lengths of the paths to discriminate the ANC$^+$ and DEC* cases (line 13).

Algorithm 3 TJ-SP implementation

```
1:  procedure AddChild(u)
2:      if u = null then
3:          return {path : [], children : 0}
4:      p ← append(copy(u.path), u.children)
5:      u.children ← u.children + 1
6:      return {path : p, children : 0}

7:  procedure Less(v1, v2)
8:      i ← 0
9:      while i < min{length(v1.path), length(v2.path)} do
10:         if v1.path[i] ≠ v2.path[i] then
11:             return v1.path[i] > v2.path[i]
12:         i ← i + 1
13:     return length(v1.path) < length(v2.path)
```

2.5 Evaluation

We empirically compare the performance of Transitive Joins with that of Known Joins. The two measurements we are interested in are the execution time overhead and the memory usage overhead incurred when using each policy. The three policy implementations we test are Known Joins using vector clocks (KJ-VC), Known Joins using snapshot sets (KJ-SS), and Transitive Joins using the spawn path algorithm (TJ-SP). The complexity bounds of Table 2.1 suggest that the jump-pointer algorithm, TJ-JP, may only pay off if the task tree is very deep. None of our benchmarks exhibit task trees deeper than 8 tasks, so we did not pursue an evaluation of TJ-JP. We chose to implement TJ-SP over TJ-GT despite the extra memory requirements because using task-local arrays instead of a shared tree of pointers can benefit from cache locality.

All policies were implemented with Armus [19] as a fallback deadlock detector. That is, if the given policy flags a join as invalid, general cycle detection is invoked to determine if the join would truly create a deadlock or if it is just a false positive.

Therefore, both the KJ and TJ verifiers are safe *and precise* as implemented. None of our benchmarks can deadlock, but because the fallback cycle detection is slow, the performance of each verifier can be impacted if the policy frequently triggers false positives.

2.5.1 Benchmark Programs

To make the comparison as fair as possible, we include the same five benchmark programs as in the KJ evaluation [22], and we have implemented our TJ verifier within the same framework as the available KJ implementation, namely, the Habanero Java language [15]. For completeness, we added a benchmark, NQueens, that is invalid under KJ but valid under TJ.

Jacobi A central finite difference stencil is iteratively computed for an 8192×8192 matrix. On each of 30 iterations, a 16×16 array of tasks is forked to compute the stencil in blocks. Each block depends on its own values from the preceding iteration, as well as values at the block boundaries for up to four neighboring blocks. Therefore, each task awaits the completion of five tasks from the previous iteration before proceeding.

Smith-Waterman Two DNA sequences, each of length 21,726, are aligned using the Smith-Waterman dynamic programming algorithm. The score array is divided into 40×40 chunks, with each chunk being computed by a task. Each task must await the completion of three neighboring tasks.

Crypt This Java Grande Forum [78] benchmark has been adapted to the Habanero Java language. The program encrypts and then decrypts 50 MB of data. Each of these two phases consists of parallel work divided among 8192 tasks that are forked and then joined by the root.

Strassen Two $n \times n$ matrices can be multiplied block-wise using seven (not eight) $(n/2) \times (n/2)$ multiplications in a divide-and-conquer strategy. At every level of recursion, the current task spawns the seven recursive multiplications and four subsequent matrix addition tasks. The recursion is cut off at blocks of size 128×128, which are multiplied directly. To multiply the top-level 4096×4096 matrices, the benchmark must spawn 30,811 tasks in a tree of depth 5. A Strassen benchmark also appears in the Cilk and Barcelona OpenMP Task Suite benchmark sets [31, 27].

Series In this Java Grande Forum [78] benchmark, one million independent tasks are spawned by the root to calculate the coefficients of the Fourier series for a simple polynomial. The computation completes once the root has joined with all tasks.

NQueens Like Strassen, NQueens employs a divide-and-conquer algorithm to allocate work among tasks. Unlike Strassen, in which each task joins with its own children or siblings, the root task of NQueens joins with all tasks in any order to collect the result. The recursion is 14 levels deep. Almost 3.4 million tasks are spawned in a tree of height 8; the 6 remaining levels of recursion proceed sequentially. An example program with the same high-level structure as NQueens was discussed in Section 2.2.3. The sequence of joins by the root task of NQueens potentially violates Known Joins, but not Transitive Joins. A divide-and-conquer NQueens also appears in the Cilk and Barcelona OpenMP Task Suite benchmark sets [31, 27].

2.5.2 Execution Time and Memory Results

We ran each benchmark program under each policy implementation on a 16-core AMD Opteron 3.2 GHz machine. The operating system was Debian 8.10; the Java version was OpenJDK 1.7, running in JDK 1.5 compatibility; the Habanero Java runtime was invoked with 16 hardware threads.

The execution time and memory overheads are presented in Table 2.2. For the

Table 2.2: Execution time and memory overheads for verification of futures. Bold face indicates best factor in each row.

Benchmark	Time (s)/ Memory (GB)	Policy Overheads KJ-VC	KJ-SS	TJ-SP
Jacobi	12.15	**1.10**×	1.12×	**1.10**×
	2.99	1.33×	1.01×	**1.00**×
SmithWaterman	5.44	1.03×	1.09×	**0.98**×
	3.47	**1.00**×	**1.00**×	**1.00**×
Crypt	0.42	9.15×	1.08×	**0.99**×
	0.31	6.55×	1.02×	**1.00**×
Strassen	7.82	**0.99**×	1.00×	1.00×
	8.39	1.03×	1.22×	**1.02**×
Series	81.10	**1.00**×	1.04×	1.01×
	0.90	1.95×	2.61×	**1.46**×
NQueens	23.55	1.48×	**1.24**×	1.32×
	5.73	1.53×	1.49×	**1.10**×
Geometric Mean Overhead	Time	1.58×	1.09×	**1.06**×
	Memory	1.73×	1.30×	**1.09**×

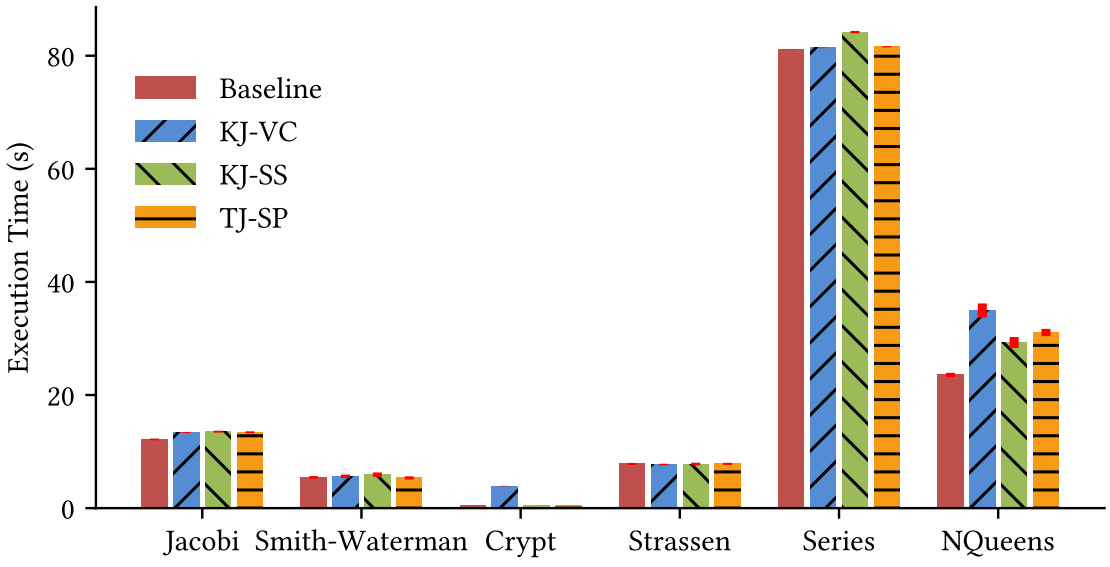

Figure 2.5: Execution times for each evaluated policy verifier, showing the mean with a 95% confidence interval.

baseline (no policy enabled), we give the absolute time in seconds and memory usage in gigabytes. The execution time is reported using the steady-state methodology [34]; we take the mean of 30 runs, after a warm-up period. The memory usage is the average amount of memory in use throughout the 30 post-warm-up runs, sampled once every 100 ms. For each of the three verifiers, we report the overhead factors for execution time and memory. Finally, we give the geometric mean overhead for each verifier across all benchmarks. The absolute execution times for the baseline and all three policies are shown with 95% confidence intervals in Fig. 2.5.

The TJ-SP time and memory overheads are no greater than a factor of 1.10× in all but two cases (memory for Series and time for NQueens). The KJ-VC and KJ-SS overheads for all the preexisting benchmarks (the verifiers were not previously evaluated on NQueens) satisfactorily replicate most of the results of Cogumbreiro et al. [22], with the exception of KJ-VC's memory usage on Jacobi, which we found to be non-competitive with the other verifiers. TJ-SP is comparable in time overhead (within 10 percentage points) to at least one of the KJ verifiers on every benchmark. On memory overhead, TJ-SP again performs comparably well to at least one of the KJ verifiers in all cases except Series and NQueens, for which TJ-SP performs significantly better than both KJ verifiers.

On Series, all three verifiers incurred significant memory overheads—at or above 1.5×. TJ-SP is the superior verifier on Series, taking three quarters as much memory as the second best, KJ-VC. Still, Series is an outlier for the memory overhead of TJ-SP, which does not exceed 1.1× on any other benchmark. The excessive memory usage for all three verifiers on Series may be due to the fact that the baseline memory footprint consists of very little data. The memory usage is thus dominated by the one million spawned tasks, since the space complexity is dependent on the number of tasks.

NQueens is the only potentially KJ-invalid program and was not previously tested

with KJ verifiers. We find that the NQueens execution time is significantly impacted by all three verifiers, with KJ-SS performing the best and KJ-VC the worst. Figure 2.5 shows a large variance in execution time for NQueens over the other benchmarks, suggesting a higher degree of nondeterminism. Recall that NQueens nondeterministically violates KJ and triggers precise cycle detection; however it never violates TJ. On memory usage for NQueens, both KJ verifiers incur a 1.5× overhead, but TJ-SP performs with only a 1.1× overhead, despite the large number of tasks.

In taking the geometric mean of execution time overheads for each verifier, we find that TJ-SP and KJ-SS have low and comparable impacts on execution time (less than 1.1×). However, the geometric mean of memory overheads reveals that the TJ verifier is the only one of the three verifiers to have a consistently low impact on memory usage (1.09×).

2.6 Related Work

2.6.1 General Approaches

Broadly, there are three categories of solutions to the deadlock problem, as outlined by Coffman et al. [18]: 1) static prevention, 2) detection at runtime, and 3) avoidance at runtime. In the first category, approaches seek to statically identify potential deadlocks or prove their absence using, for example, static analyses or type systems [87, 60, 12]. Runtime detection consists of identifying cycles of deadlocked tasks after the deadlock has already occurred [56, 46, 47, 55, 59, 82]. Finally, runtime deadlock avoidance strategies, to which this work belongs, intercept attempted join operations that will or may cause a deadlock [86, 62, 11, 13, 35, 19, 22]. The advantage of avoidance over detection is that a target program has the opportunity to recover from aborted joins; however, avoidance can be more challenging and expensive than detection [19].

A general algorithm to avoid deadlocks, not only on task termination but also on barrier synchronization, is to perform cycle detection in the *waits-for graph* to determine if each attempted join would create a deadlock cycle [71]. The advantage of cycle detection is that it is safe and precise; that is, it exactly determines whether a deadlock would arise. However, the time to verify each join with this method is quadratic in the number of tasks [71], resulting in prohibitive runtime overheads in practice [19].

2.6.2 Relationship to Known Joins

Recent work on deadlock avoidance has proposed Known Joins (KJ), a conservative policy that precludes deadlocks using a set of rules that are efficient to check at runtime [22]. KJ's advantage is that online policy verification is a low-overhead operation in practice. However, the concern with a conservative policy is that it may have limited utility by rejecting many reasonable deadlock-free programs. The implementation of a Known Joins verifier therefore uses Armus [19], a cycle-detection algorithm for deadlock avoidance, as a fallback mechanism. When a program violates KJ, Armus is invoked to precisely filter out false positives. A previous evaluation has demonstrated prohibitively expensive runtime overheads in using Armus alone, but the hybrid implementation of KJ with Armus achieves both the low overhead of a conservative policy and the precision of cycle-detection [22]. However, prior work has not established how KJ performs when a deadlock-free target program frequently triggers the fallback mechanism.

Known Joins was originally designed for the purpose of proving deadlock-freedom from data-race-freedom in the async-future model. For this reason, the KJ policy is not necessarily suited as a widely applicable deadlock-avoidance strategy. In contrast, we set out to create a new policy specifically designed to admit a large class of deadlock free programs in the async-future model. To understand this point, we ask

what interesting programs are deadlock-free but rejected by KJ?

Consider the tree formed by the forks in a program, where each newly forked task is a child, and where the forking task is the parent. KJ permits a task to join with its immediate children. But if a task joins with a descendant other than one of its immediate children, KJ requires the task to have *first* joined with all the intervening nodes on the path to the descendant. The class of behaviors we have chosen to admit in TJ is the *arbitrary descendant join*: A task should be permitted to join with any descendant regardless of what other joins have already occurred, because weakening KJ to admit arbitrary descendant joins does not compromise deadlock-freedom. This behavior arises in a natural implementation of the 'finish' construct of X10 and Habanero Java. When a finish block ends, it joins with the collection of tasks spawned transitively within the block [17]. A finish implementation may trigger a false positive under KJ unless the join order carefully respects the fork order, limiting the usefulness of KJ. We discuss this example at length in Section 2.2.3. Out novel Transitive Joins policy admits arbitrary descendant joins by transitively extending the join permission relation.

Transitivity of join permissions also has a practical advantage for the implementation of an online policy verifier. TJ retains the KJ concept of permission inheritance upon forking. But strikingly, the KJ concept that a joining (waiting) task should learn new permissions from the joinee (terminating) task becomes obsolete, as the joining task will already have these permissions by transitivity. As a consequence, a join operation does not require a TJ verifier to update any internal state, thereby simplifying a possible implementation in comparison to a KJ verifier.

2.7 Conclusion

We have presented a novel deadlock-freedom policy, Transitive Joins, for dynamic task parallelism with arbitrary join operations, which is applicable to a range of parallel

programming models such as threads and tasks with futures. The policy is a set of rules for task joins that admits only deadlock-free runs. TJ admits a strictly larger class of practical deadlock-free programs than the prior Known Joins policy.

TJ can be implemented as a runtime verifier using one of several possible algorithms, with cycle-detection as a fallback to catch false positives. The implementation of our TJ verifier for the Habanero Java language competes with and in some cases surpasses existing implementations of two KJ verifiers in minimizing both time and memory overhead. We demonstrated that TJ can efficiently verify benchmarks with a large number of tasks, incurring geometric mean time and memory overheads of 1.06× and 1.09×, respectively. This result improves over the state of the art, KJ-SS, whose time and memory overheads are 1.09× and 1.30×, respectively. Therefore, we propose that our TJ verifier is a practical choice for an always-on deadlock safety checker.

CHAPTER 3
AN OWNERSHIP POLICY AND DEADLOCK AVOIDANCE ALGORITHM FOR PROMISES

3.1 Introduction

The task-parallel programming model is based on the principle that structured parallelism (using high-level abstractions such as spawn-sync [31, 66], async-finish [17, 50, 41], futures [36, 53], barriers [66], and phasers [15, 74]) is a superior style to unstructured parallelism (using explicit low-level constructs like threads and locks). For example, deadlock freedom can be obtained by construction [74] or with efficiently verifiable policies to limit parallel constructs to deadlock-free uses [83, 22]. Structured programming also communicates programmer intent in an upfront and visible way, providing an accessible framework for reasoning about complex code by isolating and modularizing concerns. However, the *promise* construct, found in mainstream languages including C++ and Java, introduces an undesirable lack of structure into task-parallel programming. A promise generalizes a future in that it need not be bound to the return value of a specific task. Instead, any task may elect to supply the value, but the code may not clearly communicate which task is intended to do so.

Promises provide point-to-point synchronization wherein one or more tasks can await the arrival of a payload, to be produced by another task. Although the promise provides a safe abstraction for sharing data across tasks, there is no safety in the kinds of inter-task blocking dependences that can be created using promises. The inherent lack of structure in promises not only leads to deadlock-like bugs in which

Listing 3.1: A deadlock?

```
1 Promise p,q;
2 t1 = async { ... };
3 t2 = async {
4   p.get(); // stuck
5   q.set();
6 };
7 q.get();  // stuck
8 p.set();
```

tasks block indefinitely due to a cyclic dependence, but such bugs are not well-defined and are undetectable in the general case due to the lack of information about which task is supposed to fulfill which promise.

The presence of a deadlock-like cycle can only be confirmed once all tasks have terminated or blocked. For example, the Go language runtime reports a deadlock if no task is eligible to run [37]. However, if even one task remains active, there is insufficient information to prove the existence of a deadlock without further analysis. An example of such a program is in Listing 3.1; the root task and t_2 are in a deadlock that may be hidden if t_1 is a long-running task. We will explore this example in detail in Section 3.1.3. An alternative detection approach is to impose timeouts on blocking waits, which is only a heuristic solution as it involves estimating a safe upper bound for the expected wait time. A timeout that is too short can raise an alarm when there is no cycle. In both of these existing approaches, the deadlock detection mechanism may find the deadlock some time *after* the cycle has been created. It is instead more desirable to detect a cycle immediately as it forms—a distinction that earns the name "deadlock *avoidance*."

3.1.1 Promise Terminology

There is inconsistency across programming languages about what to call a promise and sometimes about what functionality "promise" refers to. The synchronization

primitive we intend to discuss is called by many names, including promise [53], handled future [65], completable future [67], and one-shot channel [25]. For us, a promise is a wrapper for a data payload that is initially absent; each *get* of the payload blocks until the first and only *set* of the payload is performed. Setting the payload may also be referred to as completing, fulfilling, or resolving the promise.

Some languages, such as C++, divide the promise construct into a pair of objects; in this case, "promise" refers only to the half with a setter method, while "future" refers to the half with a getter method. In Java, the CompletableFuture class is a promise, as it implements the Future interface and additionally provides a setter method.

Habanero-Java introduced the data-driven future [79], which is a promise with limitations on when gets may occur. When a new task is spawned, the task must declare up front which promises it intends to consume. The task does not become eligible to run until all such promises are fulfilled.

In JavaScript the code responsible for resolving a promise must be specified during construction of the promise [63]. This is a limitation that makes deadlock cycles impossible, although the responsible code may omit to resolve the promise altogether, leading to unexecuted callbacks.

Promises may provide a synchronous or an asynchronous interface. The Java concurrency library provides both, for example [67]. The synchronous interface consists of the *get* and *set* methods. The asynchronous interface associates each of the synchronous operations to a new task. A call to *supplyAsync* binds the eventual return value of a new task to the promise. The *then* operation schedules a new task to operate on a promise's value once it becomes available. The asynchronous interface can be implemented using the synchronous interface. Conversely, the synchronous interface can be implemented using continuations and an asynchronous event-driven scheduler [49]. We focus on the synchronous interface in this work.

3.1.2 Two Bug Classes

We identify two kinds of synchronization bug in which the improper use of promises causes one or more tasks to block indefinitely:

1. the *deadlock cycle*, in which tasks are mutually blocked on promises that would be set only after these tasks unblock, and

2. the *omitted set*, in which a task is blocked on a promise that no task intends to set.

However, neither of these bugs manifests in an automatically recognizable way at runtime unless every task in the program is blocked. In fact, the definitions of these bugs describe conditions which cannot generally be detected. What does it mean for no task to *intend* to set a promise? What does it mean that a task *would* set a promise once the task unblocks? In a traditional deadlock, say one involving actual locks, the cycle is explicit: Task 1 holds lock A and blocks while acquiring lock B, because task 2 is holding lock B and blocked during its acquisition of lock A. Intention to release a lock (thereby unblocking any waiters) is detectable by the fact that a task holds the lock. But we currently have no concept of a task "holding" a promise and no way to tell that a task intends to set it.

3.1.3 A Need for Ownership Semantics

Consider the small deadlock in Listing 3.1. Two promises, p, q, are created. Task t_2 waits for p prior to setting q, whereas the root task waits for q prior to setting p. Clearly a deadlock cycle arises? Not so fast. To accurately call this pattern a deadlock cycle requires knowing that task t_1 will not ever set p or q. Such a fact about what *will* not happen is generally not determinable from the present state without additional program analysis. For this reason, a deadlock cycle among promises evades runtime detection unless the cycle involves every currently executing task.

Listing 3.2: An omitted set?

```
1 Promise r,s;
2 t3 = async { // should set r,s
3   t4 = async { // should set s
4     // (forgot to set s)
5   }
6   r.set();
7 };
8 r.get();
9 s.get(); // stuck
```

Now consider the bug in Listing 3.2. Two promises, r and s, are created. According to the comments, task t_3 is responsible for setting both, and it subsequently delegates the responsibility for setting s to t_4. However, t_4 fails to perform its intended behavior and terminates without setting s. The root task then blocks on s forever. Since a bug has occurred, we would like to raise an alarm at runtime when and where it occurs. Where is this bug? Should the root task not have blocked on s? Should t_4 have set s? Should t_3 have set s? The blame cannot be attributed, and the bug may, in fact, be in any one of the tasks involved. Furthermore, *when* does this bug occur? The symptom of the bug manifests in the indefinite blocking of the root task, potentially *after* t_4 terminates successfully. If some other task may yet set s, then this bug is not yet confirmed to have occurred. Omitted sets evade runtime detection and, even once discovered, evade proper blame assignment.

We propose to augment the task creation syntax (the **async** keyword in our examples) to carry information about promise ownership and responsibility within the code itself, not in the comments. In doing so, omitted sets become detectable at runtime with blame appropriately assigned. Moreover, programmer intent is necessarily communicated in the code. Finally, in knowing which task is expected to set each promise, it becomes possible to properly discuss deadlock cycles among promises.

Listing 3.3: An omitted set in Amazon AWS SDK (v2) [48]. Code abbreviated and inlined for clarity. Comments added.

```
1  private CompletableFuture<Void> cf;
2
3  public void onComplete () {
4    ...
5    if (streamChecksumInt != computedChecksumInt)
6    {
7      onError(...); // Assumed to fulfill promise
8      return; // Do not fulfill promise again
9    }
10   ...
11   cf.complete(null); // Fulfills promise
12 }
13
14 public void onError (Throwable t) {
15   // Originally a no-op.                    // Fixed:
16                                  ⟹          cf.completeExceptionally(t);
17 }
```

3.1.4 Omitted Set in the Wild

An example of an omitted set bug was exhibited by the Amazon Web Services SDK for Java (v2) when a checksum validation failed after downloading a file [48]. An abbreviated version of the code is given in Listing 3.3; line 16 was absent prior to the bug fix. The control flow ensures that either the exception handling code or the non-exceptional code is executed, not both (line 8) [61]. However, only the non-exceptional code sets the value of a `CompletableFuture` (Java's promise) to indicate the work was completed (line 11), whereas the onError method takes no action. If the checksum validation failed, any consumer tasks, say, waiting for the file download to complete would block indefinitely. A month later the omitted set bug was identified and corrected by adding line 16 [3].

When this bug arises at runtime, the symptom (the blocked consumer) is far from the cause (the omitted set), and the bug is not readily diagnosable. If the runtime could track which tasks are responsible for which promises, then this bug could be detected and reported as an exception as soon as the responsible task terminates.

Using our approach of annotating **async** with promise ownership information, the bug would be detected when the task running the `onComplete` callback finishes, and the alarm could name the offending task and the unfulfilled promise.

3.1.5 Contributions

In this chapter we propose the addition of *ownership semantics* for promises which enables a task's intention to set a promise to be reflected in the runtime state. In so doing,

1. we enable a precise definition of a deadlocked cycle of promises in terms of runtime state;
2. we define a second kind of blocking bug, the *omitted set*, which does not involve a cycle;
3. we require important programmer intent to be encoded and to respect a runtime-verifiable policy, thereby enabling structured programming for promises.

In addition to these theoretical contributions,

1. we introduce an algorithm for avoiding the newly identifiable deadlock-cycle and omitted-set bugs by raising an alarm *when they occur*;
2. we identify properties critical for establishing the correctness of the algorithm under weak memory consistency and show how to ensure these properties hold under the TSO, Java, and C++ memory models;
3. we prove that our algorithm precisely detects every deadlock without false alarms;
4. we experimentally show that a Java implementation has low execution time and memory usage overheads on nine benchmarks relative to the original, unverified baseline (geometric mean overheads of 1.14× and 1.06×, respectively).

3.2 Defining and Implementing Ownership

In promise-based synchronization, a task does not directly await another task; it awaits a promise, thereby *indirectly* waiting on whichever task fulfills that promise. It is a runtime error to fulfill a promise twice, so their ought to be one and only one fulfilling task. However, the relationship between a promise and the task which *will* fulfill it is not explicit and inhibits the identification of deadlocks. To make this relationship explicit and meaningful, we say that each promise is *owned* by exactly one task at any given time. The owner is responsible for fulfilling the promise eventually, or else handing ownership off to another task. Ownership hand-offs may only occur at the time of spawning a new task. We augment the **async** keyword, used to spawn tasks, with a list of promises currently owned by the parent task that should be transferred to the new child.

To illustrate, we define an abstract language, showing only its synchronization instructions and leaving its sequential control flow and other instructions unspecified. For simplicity, we have abstracted away the payload values of promises and refer to individual promises by globally unique identifiers.

Definition 24. The $\mathcal{L}_p$ language consists of task-parallel programs, P, whose synchronization instructions have the syntax

$$\textbf{new}\ p \mid \textbf{set}\ p \mid \textbf{get}\ p \mid \textbf{async}\ (p_1, \ldots, p_n)\ \{P\}$$

where n may be 0.

3.2.1 Promise Semantics

We assume the usual semantics for a language with asynchronous tasks, namely, that **async** $(\ldots)$ $\{P\}$ causes a new task to be created to execute P, while the parent

task continues in parallel. When referring to the action of task t in executing an instruction, *instr*, we use the notation $t : \mathit{instr}$.

A operational semantics of promises involves a set, D, of declared promises, and a set, $F \subseteq D$ of all the currently fulfilled promises. Initially, $D = F = \emptyset$.

1. Creating a promise adds it to D. It is an error to re-create a promise.

$$\frac{\textbf{assert } p \notin D}{(D, F) \Rightarrow (D \cup \{p\}, F)} \; t : \textsf{new } p$$

2. Setting a promise adds it to F. It is an error to set a promise twice or to set a promise that has not been created.

$$\frac{\textbf{assert } p \in D \setminus F}{(D, F) \Rightarrow (D, F \cup \{p\})} \; t : \textsf{set } p$$

3. Awaiting a promise can proceed only once the promise is found to be in F. It is an error to await a promise that has not been created.

$$\frac{\textbf{assert } p \in D \quad p \in F}{(D, F) \Rightarrow (D, F)} \; t : \textsf{get } p$$

3.2.2 Ownership Semantics

We augment the preceding semantics with the concept of promise ownership, making use of the **async** annotation to indicate when promises should be transferred from one task to another.

Definition 25. The *ownership policy*, $\mathcal{P}_o$, maintains additional state during the execution of an $\mathcal{L}_p$ program in the form of a map $\mathsf{owner} : \mathit{Promise} \to \mathit{Task} \cup \{\mathit{null}\}$, initially $[\_ \mapsto \mathit{null}]$, according to the following rules. We will use the notation $t : \mathit{spawn}\ p_1 \ldots p_n\ t'$ to refer to the synthetic action of t that creates the new task,

t', via **async** $(p_1, \ldots, p_n)$. The spawn action completes *prior* to t' beginning execution. We also use the notation t : *terminate* to refer to the synthetic action of the termination of task t.

4. Upon creating a promise, the creating task is the initial owner of the promise.

$$\frac{}{\mathsf{owner} \Rightarrow \mathsf{owner}[p \mapsto t]} \; t : \textbf{new } p$$

5. When spawning a new task, the promises in the **async** annotation have their ownership transferred to the new task before it begins executing. It is an error if any of those promises is not currently owned by the parent task.

$$\frac{\textbf{assert } \forall i, \mathsf{owner}(p_i) = t}{\mathsf{owner} \Rightarrow \mathsf{owner}[p_1 \mapsto t'] \ldots [p_n \mapsto t']} \; t : \mathit{spawn}\ p_1 \ldots p_n\ t'$$

6. When a task terminates, it is an error if its set of owned promises is not empty.

$$\frac{\textbf{assert } \forall p, \mathsf{owner}(p) \neq t}{\mathsf{owner} \Rightarrow \mathsf{owner}} \; t : \mathit{terminate}$$

7. It is an error for a task to set a promise that it does not own. After setting, the promise is owned by no task.

$$\frac{\textbf{assert } \mathsf{owner}(p) = t}{\mathsf{owner} \Rightarrow \mathsf{owner}[p \mapsto \mathit{null}]} \; t : \textbf{set } p$$

8. Awaiting a promise has no affect on ownership state.

$$\frac{}{\mathsf{owner} \Rightarrow \mathsf{owner}} \; t : \textbf{get } p$$

In some task-parallel languages, like Habanero-Java, **async** automatically creates

a *future*, which can be used to retrieve the new task's return value. We can readily reproduce this behavior using promises in the pattern **new** p; **async** $(p, \ldots)$ $\{\ldots;$ **set** $p\}$.

Now that we can discuss promise ownership, we formally define the omitted set condition.

Definition 26. A program $P \in \mathcal{L}_p$ exhibits an *omitted set* if some task t has terminated and there exists a promise, p, such that $\mathsf{owner}(p) = t$.

Rule 6 detects omitted set bugs, ensuring that all promises are fulfilled upon termination, as the following theorem shows.

Theorem 27. *If $P \in \mathcal{L}_p$ exhibits an* omitted set*, then P raises an error under $\mathcal{P}_o$.*

Proof. Suppose an omitted set has occurred, with task t and promise p. Either rule 6 raises an error upon termination of t, or $\mathsf{owner}(p) \neq t$ at the time of t : *terminate*. In the latter case, no further action of the program can modify $\mathsf{owner} \Rightarrow \mathsf{owner}[p \mapsto t]$, since ownership transfer occurs only to newly spawned tasks (rule 5), precluding t as the destination. Therefore, the omitted set condition could not have actually arisen. □

3.2.3 Making Use of Ownership

Our proposed modification to the program given in Listing 3.1 is to annotate the **async** in line 3 as **async** (q), indicating that t_2 takes on the responsibility to set q. It is now possible to trace the cycle when it occurs: the root task awaits q, owned by t_2, awaiting p, owned by the root task. It is clear that t_1, whose **async** is not given any parameters, is not involved as it can set neither p nor q (rule 7).

The proposed modification to the program in Listing 3.2 is to write **async** (r, s) in line 2 and **async** (s) in line 3. That is, the information already present in the comments is incorporated into the code itself. The moment t_4 terminates, the runtime

can observe that t_4 still holds an outstanding obligation to set s. We treat this as an error immediately (rule 6), irrespective of whether any task is awaiting s.

Introducing this explicit concept of ownership into user code is minimally disruptive. It is already the case that every promise is fulfilled by at most one task, since two sets cause a runtime error. We only ask that the programmer identify this task by leveraging the existing structure of **async** directives.

For complex patterns using many promises, an object-oriented approach can reduce the burden of identifying which promises should be moved to new tasks. In our Java implementation of these language features, classes containing many promises may implement a PromiseOwner interface so that moving a composite object to a new task is equivalent to moving each of its constituent promises. A channel class is shown in Listing 3.4, illustrating that complex and versatile primitives can be built on top of promises with the aid of PromiseOwner. This class behaves like a promise that can be used repeatedly, where the nth recv operation obtains the value from the nth send operation. This behavior depends on dynamically allocated promises, and the responsibility for the sending end of the channel is associated not to the ownership of a single promise, but to the ownership of different promises at different times. It is abstraction-breaking to ask the channel user to manually specify which promise to move to a new task in order to effectively move the sending end of the channel. Instead, we give the impression that the channel object itself is movable like a promise (line 39), since it is a PromiseOwner, and the implementation of **async** relies on the ownedPromises method (line 11) to determine which promises should be moved.

3.2.4 Algorithm for Ownership Tracking

Algorithm 4 implements the $\mathcal{P}_o$ policy by providing code to be run during **new**, **async**, and **set** operations. Each promise has an owner field to store the task that is currently its owner, and each task has an associated owned list that maintains the inverse map,

Listing 3.4: Object-oriented approach to promise movement.

```
1  class Channel<T> implements PromiseOwner {
2    class Payload {
3      T value;
4      Promise<Payload> next;
5    }
6
7    Promise<Payload> producer = new Promise<>();
8    Promise<Payload> consumer = producer;
9
10   @Override // from PromiseOwner
11   Iterable<Promise<?>> ownedPromises () {
12     // Return the set of all promises that
13     // should be moved when this object moves
14     return Collections.singleton(producer);
15   }
16
17   void send (T value) {
18     // Fulfills one promise; allocates another
19     Promise<Payload> next = new Promise<>();
20     producer.set({value, next});
21     producer = next;
22   }
23
24   void stop () {
25     // Fulfills a promise
26     producer.set(null);
27   }
28
29   T recv () {
30     Payload p = consumer.get();
31     consumer = p.next;
32     return p.value;
33   }
34 }
35
36 void main () {
37   Channel<Integer> ch = new Channel<>();
38   ch.send(1);
39   async (ch) { // Move entire channel
40     ch.send(2);
41     ch.stop();
42     // No remaining promises
43   }
44   // No remaining promises
45   ch.recv(); // 1
46   ch.recv(); // 2
47 }
```

Algorithm 4 Promise Ownership Management

```
1:  procedure NEW()
2:      t ← task_cur
3:      p ← {owner : t}                          ▷ C: atomic; Java: volatile
4:      append p to t.owned
5:      return p

6:  procedure ASYNC(P, f)
7:      t ← task_cur
8:      assert p.owner = t forall p ∈ P
9:      t' ← {owned : P,
10:       waitingOn : null}                      ▷ C: atomic; Java: volatile
11:     remove all of P from t.owned
12:     p.owner ← t' forall p ∈ P
13:     do asynchronously
14:         task_cur ← t'
15:         f()
16:         assert t'.owned is empty
17:     return t'

18: procedure INIT(main)
19:     task_cur ← null
20:     ASYNC([], main)

21: procedure SET(p, v)
22:     t ← task_cur
23:     assert p.owner = t
24:     p.owner ← null
25:     remove p from t.owned
26:     set_impl(p, v)
```

owner^{-1}. The task$_{cur}$ field is a task-local value for storing the current task's identifier.

In compliance with $\mathcal{P}_o$ rule 4, the NEW procedure creates a promise owned by the currently running task (line 3) and adds this promise to that task's owned list (line 4).

ASYNC(P, f) schedules f to be called asynchronously as a new task and moves the set of promises, P, into this task. These promises are first confirmed to belong to the parent task (line 8), then moved into the child task (lines 9–12), in accordance

with rule 5. (Line 10 is in preparation for Algorithm 5, presented in Section 3.4.) Once the child task terminates, rule 6 requires that the task not own any remaining promises (line 16). The INIT procedure shows how to set up a root task to execute the main function.

Finally, SET(p, v) achieves rule 7, checking that the current task owns p and marking p as fulfilled by assigning it to no owner (lines 23–25). The procedure then invokes the underlying mechanism for actually setting the promise value to v (line 26).

As an example of how Algorithm 4 enforces compliance with $\mathcal{P}_o$, refer again to Listing 3.2. When promise s is first created, it belongs to the root task (Algorithm 4 line 4). If the **async** that creates t_4 is annotated with s, then Algorithm 4 lines 8–12 changes the owner of s to t_4. Since t_4 does not set s, upon termination of t_4, an assertion fails in Algorithm 4 line 16. The offending task, t_4, and the outstanding promise, s, are directly identifiable and can be reported in the alarm.

3.2.5 Exception Handling

In an actual implementation of Algorithm 4, some care must go into an exception handling mechanism. What code is capable of and responsible for recovering from the failed assertion in line 16? And what happens if a task terminates early, with unfulfilled promises, because of an exception?

Observe that line 16 occurs within an asynchronous task after the user-supplied code for that task has completed. One solution is to add a parameter to ASYNC so that the user can supply a post-termination exception handler, which accepts the list of unfulfilled promises, b.owned, as input. Indeed, the fix for the AWS omitted set bug included such a mechanism (not shown in Listing 3.3) [3]. Alternatively, the runtime could automatically fulfill every unfulfilled promise upon an assertion failure in line 16. Some APIs, including in C++ and Java, provide an exceptional variant of the completion mechanism for promises [53, 67]. Via this mechanism, one can

propagate an exception through the promises that were left unfulfilled, to the tasks which await them.

Finally, observe that the correctness of Algorithm 4 does not depend on reading the *contents* of a task's owned list, only on knowing when this list is empty. Therefore, the owned list could be replaced with a counter, which would reduce the memory footprint of ownership tracking. However, doing so would mean that an assertion failure in line 16 could not indicate *which* promises went unfulfilled.

3.3 Deadlock Definition with Weakly Consistent Ownership

Compliance with the $\mathcal{P}_o$ policy already eliminates omitted set bugs (Theorem 27). But the ultimate goal of $\mathcal{P}_o$ is to establish a waits-for graph that we can analyze to identify *deadlocks*.

If the map owner : $\mathit{Promise} \to \mathit{Task} \cup \{\mathit{null}\}$ were maintained in a sequentially consistent manner by serializing its accesses, we could use it to directly obtain a waits-for graph, G, over the set of tasks: while task t is executing **get** p and p is unfulfilled, G has an edge $t \to \mathsf{owner}(p)$. Deadlocks could then be identified by cycle detection in G.

In practice, however, we do not want the owner map to be a heavily synchronized data structure because that can cause contention among otherwise unrelated tasks. Instead, we will assume a weak memory model in which two tasks need not agree on the value of $\mathsf{owner}(p)$, and we will use unsynchronized accesses when possible.

We note that although the owners of different promises may be updated concurrently, it is not possible in Algorithm 4 for a write-write race to occur on the same owner field. However, we will need to ensure that some concurrent reads and writes on the same fields are ordered; we explicitly discuss the requirements in Section 3.4.2.

We can establish the correctness of our deadlock detection algorithm under weak memory models as least as strong as the following.

Definition 28. The *happens-before* (h.b.) order is the partial order over program instructions that is induced by the intra-task program order and, upon spawning each new task, the ordering of Algorithm 4 line 14 (the start of the new task) after Algorithm 4 line 12 (the last action of the parent task before spawning).

A read may observe a (not necessarily unique) last write which happens-before it or any write with which the read is not h.b. ordered. Two writes or a write and read of the same field which are not h.b. ordered are *racing*.

Lemma 29. *If w_1, w_2 are two writes to $p.\mathsf{owner}$ in Algorithm 4, then w_1 and w_2 are not racing. If r is a read of $p.\mathsf{owner}$ by task t, and r observes the value to be t, then r does not race with the write it observes.*

Proof. The two claims can be shown together. Line 3 represents the initialization of an owner field and so happens-before every other write to it. The writes in line 12 and line 24 each happen-after a read of the same field observes the value to be the currently executing task (lines 8, 23). Take this together with the fact that there are only two ways to set $p.\mathsf{owner}$ to task t: line 3, executed by t itself, or line 12, executed by the parent of t prior to spawning t. In either case, writing t to $p.\mathsf{owner}$ happens-before any read of $p.\mathsf{owner}$ by t itself. □

Since we do not assume a globally consistent state, we have to be careful in the definition of a deadlock cycle. Instead of freely referring to the owner map, we must additionally state which task's perspective is being used to view the owner map.

Definition 30. A non-empty set of tasks, T, is in a *deadlock cycle* if every task $t \in T$ is executing **get** p_t for some promise p_t, there exists a task $o_{p_t} \in T$ which observes $\mathsf{owner}(p_t) = o_{p_t}$, and T is minimal with respect to these constraints. The set of promises associated to the deadlock is $\{p_t \mid t \in T\}$.

The subtle point in this definition is that task o_{p_t} necessarily has the most up-to-date information about the owner of p_t, since o_{p_t} is itself the owner. Per Lemma 29,

Algorithm 5 Deadlock Cycle Detection

```
1:  procedure GET(p_0)
2:      t_0 ← task_cur
3:      t_0.waitingOn ← p_0                                  ▷ C: seq_cst
4:      ▷ TSO: memory fence
5:      i ← 0
6:      t_{i+1} ← p_i.owner
7:      while t_{i+1} ≠ t_0 do
8:          if t_{i+1} = null then break
9:          p_{i+1} ← t_{i+1}.waitingOn                      ▷ C: acquire
10:         if p_{i+1} = null then break
11:         if t_{i+1} ≠ p_i.owner then break
12:         i ← i + 1
13:         t_{i+1} ← p_i.owner
14:     try
15:         assert t_{i+1} ≠ t_0
16:         return get_impl(p_0)
17:     finally
18:         t_0.waitingOn ← null                             ▷ C: release
```

we know that all the writes to p_t.owner are ordered and that o_{p_t} is observing the last write, since only o_{p_t} is capable of performing the next write to follow the observed one.

3.4 Dynamic Deadlock Avoidance

We present an algorithm for deadlock avoidance that is correct under our weak memory consistency model with some additional specific consistency requirements. We will show how to attain these additional requirements in each of the TSO, Java, and C++ memory models.

3.4.1 Avoidance Algorithm

The deadlock verifier occupies the implementation of the **get** instruction, given in Algorithm 5. This approach can thereby raise an alarm in a task as soon as the task

attempts a deadlock-forming await of a promise. At the time of raising an alarm, the available diagnostic information that can be reported includes the task, the awaited promise, as well as every other task and promise in the cycle.

Upon entering GET, the currently executing task records the promise that it will be waiting on (line 3). This waitingOn field was initialized to *null* in Algorithm 4 line 10, and is always reset to *null* upon exiting GET (Algorithm 5 line 18), either normally (line 16) or abnormally (line 15). Doing so makes the algorithm robust to programs with more than one deadlock.

The goal of the avoidance algorithm is to traverse the chain formed by alternating waitingOn and owner fields. If task t is waiting on promise p, which is owned by a task t', then t is effectively waiting on whatever t' awaits. In traversing this chain, if t finds that it is transitively waiting on itself, then we have identified a deadlock (lines 7, 15). If the algorithm reaches the end of this chain without finding t again, as indicated by finding a *null* value in line 8 (p_i is already fulfilled, so it has no owner) or in line 10 (t_{i+1} is not awaiting any promise), then it is safe to commit to a blocking wait on the desired promise (line 16). Recall that p_i.owner is *null* after p_i has been fulfilled, and t_{i+1}.waitingOn is *null* when t_{i+1} is not currently executing GET.

Algorithm 5 is non-trivial because the waitingOn and owner fields may be updated concurrently with the traversal. What if stale values are read from the fields? What if an earlier link in the chain is overwritten after it is traversed? One fears these scenarios might lead to a false alarm or an undetected deadlock.

In order to guarantee that an apparent cycle always corresponds to a real deadlock, we rely on line 11 to establish that task t_{i+1} was waiting on promise p_{i+1} *while* t_{i+1} was still the owner of promise p_i. This is achieved by reading the owner field both before (lines 6, 13) and after (line 11) reading the waitingOn field (line 9). This reasoning will depend on release-acquire semantics for waitingOn, which we specify in Section 3.4.2. If the task observes the owner of p_i to have changed, it turns out that it is safe to

abandon the deadlock check and commit to the blocking wait.

In order to guarantee that every deadlock cycle is detected, we will also need to impose a little more consistency on the waitingOn field so that at least one of the tasks in the deadlock observes the most up-to-date values. We only need one alarm to prevent the cycle from forming.

3.4.2 Correctness

In our weak memory model, the correctness of Algorithm 5 depends on the following additional memory consistency requirements.

1. There is a total order, $<$, over all instances of the write in Algorithm 5 line 3, across all memory locations. Let $w_1 < w_2$. Any write preceding and including w_1 in h.b. order is visible to any read following w_2 in h.b. order.

2. The consistency of any owner field must follow from release-acquire semantics for any waitingOn field. Specifically, let w_1 be an Algorithm 4 line 3 or line 12 write to an owner field, let w_2 be an Algorithm 5 line 3 write to a waitingOn field, let r_2 be an Algorithm 5 line 9 read, and let r_1 be an Algorithm 5 line 11 read. Suppose w_1, r_1 refer to the same location, as do w_2, r_2. If w_1 happens-before w_2, if w_2 is visible to r_2, and if r_2 happens-before r_1, then w_1 is visible to r_1.

3. The write in Algorithm 5 line 18 must not become visible until the fulfillment of p_0 is visible (Algorithm 4 line 24) or it is determined that an exception should be raised (Algorithm 5 line 15).

These three additional requirements are readily attained in TSO, Java, and C++ as follows:

- Under TSO, a memory fence is needed in Algorithm 5 line 4 to achieve requirement 1 by ordering line 9 after line 3 and sequentializing all instances of line 4

with each other. TSO naturally achieves requirement 2 by respecting the local store order, as well as requirement 3 by not allowing the line 18 write to become visible early. Note that the loop body contains no fences.

- Under the Java memory model, it suffices to mark the two fields, owner and waitingOn, as volatile to satisfy all three requirements. This eliminates all write-read data races. (Remember that there are no write-write races.) The Java memory model then guarantees sequential consistency with respect to these two fields.

- In C++ both of the fields must be `std::atomic` to eliminate data races, but this alone is insufficient. Algorithm 5 line 3 must be tagged as a `std::memory_order_seq_cst` access to achieve requirement 1, establishing a total order over these writes and subsuming release consistency. Line 9 must then be tagged `std::memory_order_acquire` to achieve requirement 2. And finally, line 18 must be `std::memory_order_release` to satisfy 3.

Under the preceding requirements, it is not hard to show that Algorithm 5 raises no false alarms.

Theorem 31. *If task t fails the assertion in* GET(p) *at line 15, then a deadlock cycle exists, involving t and p.*

Proof. We have $t_0 = t$ and $p_0 = p$. If the execution had broken out of the while loop in line 8, line 10, or line 11, then the assertion would have succeeded. Therefore, it is the loop condition that fails. Upon reaching line 12 in each iteration, we have found p_i.owner to be t_{i+1} both before and after we found t_{i+1}.waitingOn to be p_{i+1}. Therefore, we know 1) that at one time t_{i+1} was the owner of p_i, and 2) that while t_{i+1} still observed itself to own p_i, t_{i+1} had invoked GET(p_{i+1}). This follows from memory consistency requirement 2. At this point in the reasoning, we do not yet know if t_{i+1} still the owner of p_i or if t_{i+1} is still awaiting p_{i+1}.

When the loop (lines 7–13) terminates with $t_{i+1} = t_0$, since t_0 is the current task, we deduce that the final t_{i+1}, set by line 6 or line 13, is the current owner of p_i. For all k modulo $i+1$, t_k at one time concurrently observed itself to be the owner of p_{k-1} and was in a call to $\textsc{Get}(p_k)$. This meets our definition of deadlock. □

The following series of lemmas builds to the theorem that Algorithm 5 detects every deadlock.

Definition 32. In a deadlock cycle comprised of tasks T, a t^* *task* is a task in T to which the line 3 write by every task in T is visible.

Lemma 33. *Every deadlock cycle has a t^* task.*

Proof. Corollary to memory consistency requirement 1. □

A t^* task, which need not be unique, should be thought of as the (or a) last task to enter the deadlock.

Lemma 34. *If a program execution exhibits a deadlock cycle comprised of tasks T and promises P, when a t^* task calls* Get *it constructs a sequence $\{t_i\}_i$ that is a subset of T and a sequence $\{p_i\}_i$ that is a subset of P.*

Proof. We have $t_0 = t^* \in T$ and, by definition, $p_0 \in P$. If the loop immediately terminates, then $t_1 = t_0 \in T$, and we are done. Otherwise, the values of t_{i+1} and p_{i+1} inductively depend on t_i and p_i. By definition of deadlock, one of the tasks in T, call it o_{p_i}, observes itself to be the owner of p_i. The most recent write to p_i.owner (recall all the writes are ordered by Lemma 29) occurred in program order before o_{p_i}'s line 3 write. Therefore, memory consistency requirement 1 establishes that t^* must read $t_{i+1} = o_{p_i} \in T$ in line 11. By definition of t^* and by memory consistency requirement 3, we see that line 9 observes t_{i+1}'s line 3 write, not its line 18 write. Thus, $p_{i+1} \in P$ by definition of deadlock. □

Lemma 35. *If a program execution exhibits a deadlock cycle comprised of tasks T, no t^* task executes a diverging loop (lines 7–13) in its call to* GET.

Proof. Suppose, during the call to GET by t^*, the loop does not terminate. Thus $t_i \neq t_0$ for any $i > 0$. But by Lemma 34, the infinite sequence $\{t_i\}_i$ is a subset of T. Therefore, T, in fact, exhibits a smaller cycle not involving t_0, violating the minimality condition in the definition of deadlock cycle. □

Theorem 36. *If a program execution exhibits a deadlock cycle comprised of tasks T and promises P, at least one task in T fails the assertion in Algorithm 5 line 15.*

Proof. Suppose for the sake of contradiction that a deadlock cycle arises and yet no assertion fails. So every task $t \in T$ enters the GET procedure and either blocks at line 16 on a promise in P or diverges in an infinite loop.

No task exits the loop by failing the loop condition, $t_{i+1} \neq t_0$, since this would directly fail the assertion in line 15.

For each invocation of GET by a t^* task, the loop cannot break in line 8 or line 10 because Lemma 34 implies no tasks or promises in the sequence are *null*. If the loop breaks in line 11, then t^* has observed the owner of p_i to change from one read to the next. This is impossible: both reads observe the current owner, o_{p_i}, by the same reasoning as in the proof of Lemma 34. Finally, the loop cannot diverge for t^*, by Lemma 35. Since there exists at least one t^* task, by Lemma 33, we have a contradiction. □

Corollary 37 (to Theorems 31, 36)**.** *Algorithm 5 is precise and correct, guaranteeing the existence of a deadlock when an alarm is raised and raising an alarm upon every deadlock.*

3.5 Evaluation

We evaluate the execution time and memory usage overheads introduced by our promise deadlock avoidance algorithm on eight task-parallel programs. The overheads are measured relative to the original, unverified baseline versions.

1. Conway [85] parallelizes a 2D cellular automaton by dividing the grid into chunks. We adapted the code from C to Java, using our Channel class (Listing 3.4) in place of MPI primitives used by worker tasks to exchange chunk borders with their neighbors.

2. Heat [14] simulates diffusion on a one-dimensional surface, with 50 tasks operating on chunks of 40,000 cells for 5000 iterations. Neighboring tasks again use Channel in place of MPI primitives.

3. QSort sorts 1M integers using a parallelized divide-and-conquer recursion; the partition phase is not parallelized. This is a standard technique for parallelizing Quicksort [30] and has been previously implemented using the Habanero-Java Library [50]. We implemented the finish construct, which awaits task termination, using promises.

4. Sieve counts the primes below 100,000 with a pipeline of tasks, each filtering out the multiples of an earlier prime. A similar program is found in prior work [64].

5. SmithWaterman (adapted from HClib [41]; also used in prior work [83, 22]) aligns DNA sequences having 18,000-20,000 bases. Each task operates on a 25×25 tile.

6. Strassen (such a program is found in the Cilk, BOTS, and KASTORS suites [31, 27, 81]) multiplies sparse 128×128 matrices containing around 8000 values.

Divide-and-conquer recursion issues asynchronous addition and multiplication tasks, up to depth 5.

7. StreamCluster (from PARSEC [8]) computes a streaming k-means clustering of 102,400 points in 128 dimensions, using 8 worker tasks at a time. We replaced the OpenMP barriers with promises in an all-to-all dependence pattern.

8. StreamCluster2 reduces synchronization in StreamCluster by replacing some of the all-to-all patterns with all-to-one when it is correct to do so. We also correct a data race in the original implementation.

All benchmarks were run on a Linux machine with a 16-core AMD Opteron processor under the OpenJDK 11 VM with a 1 GB memory limit. A thread pool schedules asynchronous tasks by spawning a new thread for a new task when all existing threads are in use. This execution strategy is necessary in general for promises because there is no *a priori* bound on the number of tasks which can block simultaneously. We measured both execution time and, in a separate run, average memory usage by sampling every 10 ms. Each measurement is averaged over thirty runs within the same VM instance, after five discarded warm-up runs; this is a standard technique to mitigate the variability of JVM overheads, including JIT compilation [34].

Table 3.1 gives the unverified baseline measurements for each program and the overhead factors introduced by the verifiers. The table also gives the geometric mean of overheads across all benchmarks. There is an overall factor of 1.14× in execution time and 1.06× in memory usage. Table 3.2 gives additional information useful for understanding the intensiveness of the synchronization in each benchmark. The total number of tasks and the average number of gets and sets per millisecond are reported. Figure 3.5 represents the execution times of each benchmark, showing the 95% confidence interval. The low overheads indicate that our deadlock detection algorithm does not introduce serialization bottlenecks.

Table 3.1: Execution time and memory overheads for promise ownership and deadlock avoidance.

	Time		Memory	
Benchmark	Baseline (s)	Overhead	Baseline (MB)	Overhead
Conway	4.43	1.01×	314.06	0.98×
Heat	5.06	1.00×	51.28	1.00×
QSort	3.14	0.98×	115.92	1.08×
Sieve	1.24	2.07×	140.39	1.18×
SmithWaterman	4.26	1.10×	444.44	1.40×
Strassen	0.58	1.04×	116.69	1.00×
StreamCluster	14.48	1.19×	91.02	0.95×
StreamCluster2	16.81	0.99×	89.96	0.99×
Geometric Mean Overhead		1.14×		1.06×

Table 3.2: Baseline promise benchmark metrics.

Benchmark	Tasks	Gets/ms	Sets/ms
Conway	101	361.74	361.58
Heat	51	98.92	98.89
QSort	786 035	250.13	250.12
Sieve	9 594	37 285.39	74 547.63
SmithWaterman	569 857	536.08	401.53
Strassen	58 998	102.20	544.11
StreamCluster	33	39.27	274.89
StreamCluster2	33	17.92	125.93

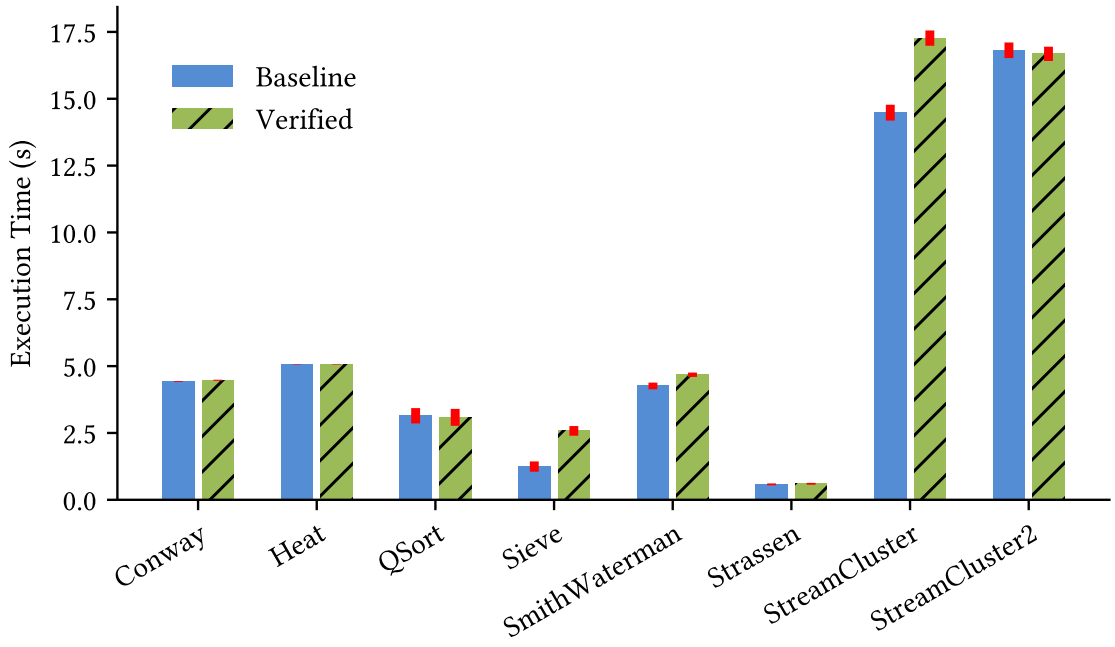

Figure 3.5: Execution times for promise benchmarks with ownership semantics and deadlock avoidance showing the mean with a 95% confidence interval (red).

The overall execution time overheads are within 1.1× for Conway, Heat, QSort, SmithWaterman, Strassen, and StreamCluster2. The same is true of the memory overheads for this subset of benchmarks, excepting SmithWaterman.

It is worth noting that the execution overhead for Sieve is in excess of 2×. Sieve has the single highest rate of get operations by an order of magnitude (over 37,000, compared to SmithWaterman's 536). The Sieve program requires almost 9594 tasks to be live simultaneously, each waiting on the next, with the potential to form very long waits-for chains for Algorithm 5 to traverse.

We can also remark on the 1.4× memory overhead in SmithWaterman. Unlike Conway, Heat, Sieve, and both StreamCluster benchmarks, in which most promises are allocated by the same task that fulfills them, SmithWaterman allocates all the promises in the root task and moves them later. In maintaining the owned lists in Algorithm 4, one can make trade-offs between speed and space. Our implementation

favors speed. So instead of literally removing a promise p from t.owned in line 11 and line 25, we simply rely on the fact that p.owner is no longer t to detect that p should no longer be counted in line 16.

For comparison with deadlock verification in other settings, the Armus tool [19] can identify barrier deadlocks as soon as they occur, with execution overheads of up to 1.5× on Java benchmarks. Our benchmark results represent an acceptable performance overhead when one desires runtime-identifiable deadlocks and omitted sets with attributable blame.

3.6 Related Work

Task-parallel programming is prevalent in a variety of languages and libraries. Multilisp [43] is one of the earliest languages with futures, a mechanism for parallel execution of functional code. Fork-join parallelism is employed in Cilk [31], and the more general async-finish with futures model was introduced in X10 [17]. Habanero-Java [15] modernized X10 as an extension to Java and, later, as a Java library, HJlib [50]; this language incorporates additional synchronization primitives, such as the phaser [74] and the data-driven future [79], which is a promise-like mechanism. Many other languages, libraries, and extensions include spawning and synchronizing facilities, whether for threads or lightweight tasks, including Chapel [16], Fortress [4], OpenMP [66], Intel Threading Building Blocks [51], Java [36], C++17 [53], and Scala [42]. JavaScript, though a single-threaded language, still uses an asynchronous task model to schedule callbacks on an event loop [58].

The promise, as we define it, can be traced back to the I-structures of the Id language [5], which are also susceptible to deadlock. Cells of data in an I-structure are uninitialized when allocated, may be written to at most once, and support a read operation that blocks until the data is available.

The classic definition of a deadlock is found in Isloor and Marsland [52], which

is primarily concerned with concurrent allocation of limited resources. Solutions in this domain fall into the three categories of Coffman: static prevention, run-time detection, and run-time avoidance [18].

We consider logical deadlocks, which are distinct from resource deadlocks in that there is an unresolvable cyclic dependence among computational results. Solutions in the logical deadlock domain include techniques that dynamically detect cycles [59, 56, 55, 46, 82, 47], that raise alarms upon the formation or possible formation of cycles [2, 11, 35, 19, 22, 83], that statically check for cycles through analysis [87, 60, 64] or through type systems [12, 80], or that preclude cycles by carefully limiting the blocking synchronization semantics available to the programmer, either statically or dynamically [17, 74, 15, 22, 83]. The present work includes a dynamic, precise deadlock avoidance algorithm, enabled only by the introduction of a structured ownership semantics on the otherwise unrestricted promise primitive.

Futures are a special case of promises where each one is bound to a task whose return value is automatically put into the promise. Transitive Joins [83] and its predecessor, Known Joins [22], are policies with runtime algorithms for deadlock avoidance on futures. They are, in general, not applicable to promises. These two techniques impose additional structure on the synchronization pattern by limiting the set of futures that a given task may await at any given time.

Recent work identifies the superior flexibility of promises over futures with the problematic loss of a guarantee that they will be fulfilled and develops a *forward* construct as a middle-ground [28]. Forwarding can be viewed in terms of delegating promise ownership, but it is restricted in that 1) it moves only a single promise into a new task, and 2) in particular, it moves only the implicit promise that is used to retrieve a task's return value. In terms of futures, forwarding amounts to re-binding a future to new task.

Other synchronization constructs benefit from similar annotations to the one we

have proposed for promises. For example, the MPI blocking receive primitive must name the sending task; from this information a waits-for graph for deadlock detection can be directly constructed [46]. Moreover, languages with barriers and phasers sometimes require the participating tasks to *register* with the construct [74]. Notably, this kind of registration is absent from the Java API, which is problematic for the Armus deadlock tool [19]. In that work, registration annotations had to be added to the Java benchmarks in order to apply the Armus methodology.

In this work we considered programs which only use promises for blocking synchronization, and we constrained ownership transfer to occur only when a task is spawned. Since a promise can have multiple readers or no readers at all, it is not possible in principle to use one promise to synchronize the ownership hand-off of a second promise between two existing tasks. We cannot guarantee that the receiving task exists and is unique. In future work, one could consider a slightly higher abstraction in the form of a pair of promises acting like a rendezvous, which is a primitive in languages like Ada and Concurrent C [33]. Such a synchronization pattern could be leveraged to hand off promise ownership since there would be a guaranteed single receiving task.

The Rust language incorporates affine types in its move semantics to ensure that certain objects have at most one extant reference at all times [72]. The movement of promise ownership from one task to another and the obligation to fulfill each promise exactly once may be expressible at compile time through the use of a linear type system, which restricts references to exactly one instance.

3.7 Conclusion

We have introduced an ownership semantics for promises, whereby each task is responsible for ensuring that all of its owned promises are fulfilled. This mechanism makes it possible to identify a bug, called the omitted set, at runtime when the bug

actually occurs, reporting which task is to blame. Prior to this work, such bugs could only be discovered after the fact when an awaiting task blocks indefinitely or times out. The ownership mechanism also makes it meaningful, for the first time, to formally define, discuss, and detect deadlock cycles among tasks synchronizing with promises. Such a bug is now detectable as soon as the cycle forms.

In our approach, code which spawns a new task is required to name the promises which are to be transferred to the new task. The programmer must already be aware of this critical information in order to even informally reason about omitted set and deadlock bugs. We now ask that it be explicitly notated in the code.

We provided an algorithm for checking compliance with the ownership policy at runtime, which detects omitted sets, and an algorithm for avoiding deadlock cycles. Both types of bug are detected when they occur, not after-the-fact. Our deadlock verifier is provably precise and correct under a weak memory model and we described how to obtain this correct behavior under the TSO, Java, and C++ memory models. Every alarm corresponds to a true deadlock and every deadlock results in an alarm. Experimental evaluation demonstrates that our lock-free approach to deadlock avoidance exhibits low execution time and memory overheads relative to an uninstrumented baseline.

CHAPTER 4
STRUCTURED USE OF PROMISES WITH SAFE, APPROXIMATE DEADLOCK-FREEDOM POLICIES

4.1 Introduction

We have seen that promises, which provide single-use point-to-point coordinating, are an unstructured mechanism for synchronizing task-parallel programs and, hence, are subject to deadlock bugs. However, these bugs can only be detected and properly reasoned about by first imposing additional structure on promises. In Chapter 3 we defined promise ownership and a policy, $\mathcal{P}_o$, for ensuring every promise is fulfilled by its owning task [84].

But there is a still a penalty caused by the inherent lack of structure in promises that persists even in deadlock-free code. A promise's set and get operations intertwine the concerns of two regions of code which need not lie in the same function, and, indeed, do not lie in the same task. This unstructured inter-task communication is problematic in the same sense that goto statements, which encode unstructured intra-task control flow, are problematic and discouraged in favor of nested blocks that form if-then-else statements and loops [26].

In this chapter we impose further structure onto promises in way that still allows their convenient and flexible use, but precludes deadlocks and enforces structured patterns. Our first contribution is a run-time policy, $\mathcal{P}_{oc}$, for efficiently accepting or rejecting unstructured get operations on promises that may form deadlock cycles. We define $\mathcal{P}_{oc}$ to raise an alarm when there is concurrent bi-directional synchronization

Listing 4.1: Nondeterministic deadlock exhibiting a potentially long and confusing cycle.

```
1 Promise [] A = new Promise[I]; // initialized
2 for (int i = 0; i < I; i++) {
3   async (A[i]) {
4     A[i].set(...); // initial data
5     while (...) {
6       // Replace old value with new promise.
7       double v = A[i].get();
8       A[i] = new Promise();
9       // Await 4 neighbors.
10      v += A[(i-2)%I].get();
11      v += A[(i-1)%I].get();
12      v += A[(i+1)%I].get();
13      v += A[(i+2)%I].get();
14      // Release new value.
15      A[i].set(v/5);
16    }
17  }
18 }
```

between groups of tasks that is indicative of a potential deadlock. We prove that $\mathcal{P}_{oc}$ identifies all actual deadlocks among promises. Our implementation of a runtime verifier for this policy is competitive with cycle detection on a suite of benchmarks.

Since $\mathcal{P}_{oc}$ over-approximates the conditions for deadlock, it can reject some desirable deadlock-free programs. Therefore, we propose a novel pseudo-synchronization feature called the *guard* block, which serves both as a structured mechanism for the programmer to communicate intent in complicated synchronization patterns and as a way to suppress alarms on deadlock-free code, *without* compromising the correctness of the policy and *without* losing any parallelism by introducing unnecessary synchronization. That is, the misuse of guards cannot create a deadlock that escapes detection by the verifier, and guards never add any waits-for dependences to the computation graph.

4.1.1 Motivating Example

Listing 4.1 demonstrates a buggy use of promises in an iterative averaging algorithm that uses a 5-point stencil. A collection of I worker tasks is spawned (line 3) to compute the values of an array over a number of rounds. Each cell is repeatedly updated to be the average of itself and two neighbors on either side (lines 7–15). However, this code exhibits a race wherein the promise for the new value can be written over the promise for the old value (line 8) prior to the dependent neighbors' obtaining the old value. Let us assume that the memory model may allow such a write to become visible without synchronization.[1] As a result, task 0 could accidentally block (line 12) on a promise to be fulfilled by task 1 (line 15) in the *same round*, which could in turn block (line 11) on the promise yet to be fulfilled by task 0 (line 15), forming a cycle. But the situation is far worse: Much longer and convoluted cycles can arise along the array of promises. For example, a cycle on indices $0 \to 2 \to 4 \to 3 \to 1 \to 0$ can form since dependences exist between neighbors that are one or two cells away in either direction.

As a red herring, notice that the array is accessed modulo I, so that cells on the far left are neighbors to the cells on the far right. Upon finding a runtime cycle such as $0 \to I-2 \to I-4 \to \ldots \to 2 \to 0$, one might suspect that the inherit cyclicity of the neighbor relation created by the modulo-I accesses is to blame; but it is not. If the race were corrected by lowering line 8 after line 13, then no deadlock cycles would be possible.

A precise cycle detection algorithm applied to the waits-for graph (like Algorithm 5) can report any cycle that arises, but it cannot provide assistance in identifying blame. It can report the final cycle-forming get, but that is nondeterministic and could be any one of the edges in the cycle. Or it can report the entire cycle, which,

[1] Racy behaviors are not necessary to create deadlocks with promises, as trivial examples can show. But we find the complexity of this example compelling.

in this case, may be an overwhelming amount of information with no indication as to where to begin debugging.

We propose a deadlock-freedom policy, $\mathcal{P}_{oc}$, extending $\mathcal{P}_o$ from Definition 25, that reports the problematic waits and the tasks involved and, additionally, higher-level information about why the behavior is problematic. This higher-level information takes the form of two or three representative sibling tasks, which are directly, or indirectly via their children, synchronizing in a potentially deadlocking manner. These representative tasks are the ancestors of, if not equal to, the offending tasks which triggered the alarm. For the above scenario in Listing 4.1, our policy discovers when there are two concurrent waits-for edges in opposite directions, one going from task i to task j and one going from task j to task i', where $i, i' > j$ (using the indices in the for loop). This key property is present in every cycle that can arise in Listing 4.1. $\mathcal{P}_{oc}$ raises an alarm once the second of these two conflicting awaits is initiated, even if no true cycle exists. Importantly, the alarm can identify both offending waits; indeed, there is not just one offender, since the same alarm can be nondeterministically raised upon either wait, and the deadlock, if it exists, involves both waits.

The governing principle of $\mathcal{P}_{oc}$ is that at no time should there concurrently arise an await from task a to task b and from task b to task c in which tasks a and c are *both* ordered before b. The ordering we use is the same one employed by Transitive Joins in Chapter 2, which is based on lowest common ancestors. The $\mathcal{P}_{oc}$ policy for promises is inherently more difficult to verify than Transitive Joins for futures because $\mathcal{P}_{oc}$ rejects two awaits if they occur concurrently, when either action occurring alone would be acceptable, whereas Transitive Joins accepts or rejects single awaits individually.

4.1.2 Outline

We briefly recall Chapter 3 to discuss how promise ownership allows us to meaningfully discuss deadlocks as waits-for cycles among tasks and detect such cycles using

a precise algorithm (Section 4.2). We then employ ownership in defining our novel conservative deadlock-freedom policy, $\mathcal{P}_{oc}$, in Section 4.3. To address the inconveniences of false alarms under $\mathcal{P}_{oc}$, we introduce a novel primitive, the *guard block*, and describe how $\mathcal{P}_{oc}$ can be relaxed by these guards to form a more flexible policy, $\mathcal{P}_{ocg}$, which is still deadlock-free (Section 4.4).

We provide algorithms for dynamically verifying $\mathcal{P}_{ocg}$, of which $\mathcal{P}_{oc}$ is a special case (Section 4.5), and empirically compare an implementation against a precise cycle detector (Section 4.6). A discussion of prior work relating to promises and deadlock avoidance can be found in the previous chapter, in Section 3.6. We summarize the elements of our contribution in Section 4.7.

4.2 Precise Cycle Detection

One solution to the deadlock problem for promises is to perform precise cycle detection on the waits-for graph, as developed in Chapter 3 in Algorithm 5. Since a task can only await one promise at a time in $\mathcal{L}_p$, the waits-for graph has out-degree at most one. Every time a task issues an await, traversing this graph for cycle detection takes time proportional to the length of the newly formed cycle if it exists, or the length of the dependence chain beginning at the current task if there is no cycle. We saw that the algorithm to perform this traversal is non-trivial because dependences may be added to or removed from the graph by other tasks during the computation.

Precise cycle detection has a few theoretical flaws. First, in the worst case, the traversal covers every task in the program, and very many tasks may be encountered even in the common case that there is no cycle. Second, the number of tasks in a given dependence chain is not fixed but can grow even while a cycle detection is in progress. Moreover, the number of tasks is not even bounded, since new tasks can spawn and enter the dependence chain. Thus, the amount of computation required to verify that no cycle will be created by the addition of a single await is not bounded.

Listing 4.2: Pathological example for precise cycle detection.

```
1 Promise p = new Promise();
2 while (...) {
3   Promise q = new Promise();
4   async (p) {
5     q.get();
6     p.set();
7   }
8   p = q;
9 }
10 p.set();
```

Not even once the computation begins does the computation become bounded. This opens the possibility of processor time being wastefully applied to long verifications instead of user-code computation.

A concrete example of schedule-dependent wasteful computation is demonstrated by Listing 4.2. This program repeatedly spawns tasks that enter into a long chain of waits-for dependences. There exists a schedule for the program which is trivial for a cycle detector to verify: If the tasks execute one at a time, in reverse order, after line 10, then all the waits-for chains have length zero. Each promise is already fulfilled by the time it is awaited, and verification is instantaneous.

However, there also exists a schedule for this program which triggers the pathological behavior of the cycle detector. Let N be the total number of iterations. If the tasks execute one at a time, in reverse order, right *before* line 10, then the verification of line 5 in the task from iteration i requires traversing $N - i$ waits-for dependences. In total, the number of dependences traversed in calls to the verifier is quadratic in N.

Precise cycle detection also has a practical flaw in that it can only provide two pieces of information when a cycle is found, neither of which is very useful. First, an alarm reports one of the final, cycle-forming awaits; however, any one of the awaits in the cycle could have nondeterministically been the final one to arrive. (It is not

possible for a happens-before relation to exist between any pair of arrivals, as this proves the absence of a deadlock.) Moreover, "the" final await need not be unique, and a single cycle can trigger multiple alarms if multiple awaits arrive concurrently. The second piece of information an alarm can report is the list of every promise and task in the entire cycle. However, cycles can contain arbitrarily many awaits, whereas all but one of those awaits may be correct and bug-free.

4.3 Approximate Cycle Detection

As an alternative to precise cycle detection, we propose that every time a task awaits a promise, a fast, approximate check should be executed to determine if this blocking call risks creating a deadlock cycle. This check should be correct in that it always identifies the formation of a real cycle, but we allow the check to be conservative. We present such a check that does not have the theoretical flaws of precise cycle detection that we discussed and additionally provides more meaningful alarms.

We can reduce the worst-case complexity of identifying cycles via two approximations. The first is to project waits-for edges to lowest common ancestors (LCAs) in the task tree (Section 4.3.1), and the second is to look for a local feature at the LCA, which we call a *concave turn*, that is a necessary condition for a cycle (Section 4.3.2).

The LCA projection has a simple $\mathcal{O}(h)$ algorithm, where h is the height of the task tree *at the time of initiating* the check.[2] Whereas continued forking of tasks can stall an in-progress waits-for graph traversal, it cannot affect an LCA computation. LCA projection does not require any synchronization because the underlying tree data structure we use grows monotonically and is immutable. A concave turn requires only constant time to identify, once the LCA computation has been performed, and it has a lock-free algorithm that uses atomic integer operations.

[2]We know from Chapter 2 that algorithms with better worst-case complexity exist for the same computation [45], but have higher overheads and space requirements. Since h is often very small, we employ the simple algorithm.

4.3.1 Projection to LCAs

We are able to leverage the lowest common ancestor of vertices to extract important information from a potential cycle and localize this information to the vicinity of a single node for easier analysis. The LCA may be thought of as an approximating representative for a pair or a larger collection of vertices. More precise information than can be encoded in a *single* ancestor vertex is retained if we also examine the (at most two) children of the LCA which lie on the paths toward each of the original vertices. For this chapter, we use a modified definition of the lca^+ function that was introduced in Chapter 2. Here, we want the function to always return an *edge* that encodes the desired information.

Definition 38. If a, b are vertices in a rooted tree, the *extended lowest common ancestor* is the triple (c, a^*, b^*) such that

1. c is the lowest common ancestor of a and b;
2. if $a = c$ then $a^* = a$; otherwise a^* is the unique child of c that is an ancestor of or equal to a;
3. if $b = c$ then $b^* = b$; otherwise b^* is the unique child of c that is an ancestor of or equal to b.

The *LCA projection* map lca^+ sends $(a, b) \mapsto (a^*, b^*)$.

It is safe to look for deadlock cycles in the promise waits-for graph *after* projecting the graph under lca^+. We show this by characterizing what the projection does to cycles through a series of lemmas.

Definition 39. The definition of lowest common ancestor generalizes to a set of vertices, V, with $|V| \geq 2$, as follows: $lca(V)$ is the singleton vertex in the fixed point of $V \mapsto \{lca(x, y) \mid x, y \in V, x \neq y\}$. The fixed point exists because, while $|V| \geq 2$, the height of the lowest vertex in V is strictly increasing.

Definition 40. If c is a vertex in a tree, T, the *vicinity* of c, denoted $A(c)$ is the set of vertices comprised of c and the children of c.

First, some portion of a connected subgraph is projected into the vicinity of the LCA of the whole subgraph.

Lemma 41. *Let $G = (V, E)$ be a directed graph with a weakly connected non-trivial subgraph, H, and T a rooted tree over V. Let V_H be the vertex set of H, and let $c = lca(V)$ with respect to T. Put $H^* = (V, \{lca^+(a, b) \mid (a, b) \in E\})$. There exists an edge $(u, v) \in H^*$ such that $u, v \in A(c)$.*

Proof. The result is trivial if H contains any edge incident on c. In the remaining case, there must exist two *distinct* children a and b of c, such that H contains an edge incident on the subtree rooted at a and an edge incident on the subtree rooted at b. For if not, then the vertices of H are entirely contained in a single subtree, meaning that c is not, in fact, the LCA. Moreover, since H is connected, H contains an edge, (x, y), incident on *both* the subtree rooted at a and the subtree rooted at b. By definition, $lca^+(x, y)$ must be (a, b) or (b, a), and $a, b \in A(c)$, as desired. □

Second, we show that the image of a cycle under projection still contains a cycle, which is, moreover, located in the vicinity of the LCA.

Lemma 42. *Let $G = (V, E)$ be any directed graph, and T a rooted tree over the same vertices. Put $G^* = (V, \{lca^+(a, b) \mid (a, b) \in E\})$, where the LCA is computed with respect to T. Suppose G contains a cycle, C, whose vertex set is V_C. Let $c = lca(V_C)$. Then G^* has a cycle whose vertices are contained in $A(c)$.*

Proof. Let $C^* \subseteq G$ be the subgraph of $\{lca^+(x, y) \mid (x, y) \in C\}$ whose vertices lie in $A(c)$. It remains to show that C^* contains a cycle.

Since C is connected, we know that C^* is not empty by Lemma 41. For the sake of contradiction, assume that C^* contains no cycle. Since C^* is not empty, let $v \in C^*$

be chosen such that v has no out-edge in C^*. Since v must be incident on some edge in C^*, there exists some in-edge $(a, v) \in C^*$. There also exists some $(x, u) \in C$ such that $lca^+(x, u) = (a, v)$.

- Suppose v is a child of c, and let S be the set of vertices in the subtree of T rooted at v. By Definition 38, u is a descendant of or equal to v, so $u \in S$. Let w be the unique vertex such that $(u, w) \in C$. It follows that $w \in S$ as well, since otherwise $lca^+(u, w)$ would be an out-edge for v in C^*. Repeat this process inductively to construct the maximal path $Q \subseteq C$ out of u. Hence all vertices of Q are in S. Since C is a cycle and Q is maximal, then $Q = C$, so all vertices of C are in S. Therefore, the LCA of V_C is, in fact, v, the root of subtree S, not c. This is a contradiction.

- Suppose $v = c$. Since c is the LCA of V_C, it must be that $u = v = c \in C$. Let $w \in C$ be chosen so that $(v, w) \in C$. Then $lca^+(v, w) = (v, b)$ where b is some child of c. Therefore, v does, in fact, have an out-edge $(v, b) \in C^*$, contradicting the assumption.

□

Now we know that the special cycle, C^*, in Lemma 42 lies entirely in the vicinity of its LCA. (Strictly speaking, C^* may be a collection of disjoint cycles.) Finally, we show that this is true of *every* cycle arising in G^*.

Lemma 43. *Let G^* be defined as in Lemma 42, let C be any cycle in G^*, and let V_C be the vertex set of C. It is the case that $V_C \subseteq A(lca(V_C))$.*

Proof. Observe that lca^+ is an idempotent map. Therefore, by applying Lemma 42 on the graph C, which is already an image under lca^+, we obtain a corresponding C^* with the desired property, and find that $C = C^*$. □

The ultimate result of LCA projection on cycles, then, is that large cycles are reduced to a collection of smaller cycles, each lying within the vicinity of some vertex.

4.3.2 No Concave Turns

After LCA projection, cycles in the waits-for graph become smaller and localized. However, such cycles can still be very long since one task can have arbitrarily many children. To avoid traversing all the children, which may be spawning concurrently, we will detect it via a local feature that is a necessary condition for a cycle.

A first approach one might take is to impose an ordering over tasks and look for waits-for dependences that are not aligned with the ordering. In this work, we will use the same preorder traversal of the task tree that we used in Transitive Joins in Chapter 2, which we will denote $<_{TJ}$. Recall that siblings, a and b, have $a <_{TJ} b$ if a was spawned more recently than b, that a parent is ordered before its children, and that the ordering is transitive.

However, directly imposing $<_{TJ}$ (or any fixed ordering) is overly restrictive. The benefit of using promises, as opposed to a weaker construct like futures, is that the programmer can construct waits-for dependences in either direction between a pair of tasks. For any two tasks, $a <_{TJ} b$, a policy like Transitive Joins fixes the dependence order up front to allow only a to await b, but never b to await a. Promises would become no more powerful than futures if we subject the dependences to a fixed ordering in this way.

A more flexible approach is to impose *monotonicity* on waits-for chains: allow any given chain to run entirely *with* the task ordering or entirely *against* the task ordering. For our purposes, it turns out even monotonicity is still too restrictive. For example, monotonicity would forbid the very reasonable `convex` function in Listing 4.3, where a parent is both awaiting a child (lines 3, 6) and being waited on by another child (lines 4, 7) in order to coordinate the two of them. (This pattern arises, for example, in the StreamCluster benchmark that we use in the evaluation.)

A still more flexible approach, which is the approach we take, is to forbid concavity in waits-for chains with respect to the task ordering.

Listing 4.3: Examples of convex and concave turns.

```
1 // No alarm
2 void convex (Promise x, Promise y) {
3   async (y) { y.set(); }
4   async { x.get(); }
5   // Parent mediates children
6   y.get();
7   x.set();
8 }
9
10 // Possible false alarm
11 void concave (Promise x, Promise y) {
12   async (x) {
13     // Child mediates parent and younger sibling
14     y.get();
15     x.set();
16   }
17   async (y) { y.set(); }
18   x.get();
19 }
```

Definition 44. A *concave turn* is a waits-for chain $a \to b \to c$ where $a \leq_{TJ} b$ and $c \leq_{TJ} b$. A *convex turn*, similarly, has $b \leq_{TJ} a$ and $b \leq_{TJ}$.

Lemma 45. *Every cycle* $\{a_i \to a_{i+1 \bmod n}\}_{i=0}^{n-1}$ *of any length, including* $n = 1$*, exhibits at least one concave turn.*

Proof. Choose i so that $a_i \geq_{TJ} a_j$ for all j. Then we have $a_{i-1 \bmod n} \to a_i \to a_{i+1 \bmod n}$, but $a_{i-1 \bmod n} \leq_{TJ} a_i$ and $a_{i+1 \bmod n} \leq_{TJ} a_i$, which is a concave turn. □

The asymmetry of forbidding concave turns requires some justification. Is no-convex-turns just as good of a policy? The symmetry breaker is that parents are ordered before their children, and it is more natural to let the parent perform the role of coordination; more generally, we let syntactically later code coordinate interactions between earlier code. We can see the contrast in Listing 4.3. In convex, the two children (lines 3, 4) are effectively communicating with each other, but the synchronization passes through their parent (lines 6, 7). We allow this pattern. However, in concave, the older child (line 12) performs the coordination before the reader has seen

the tasks that are being coordinated. We disallow this pattern, even though there is not necessarily a deadlock, and justify disallowing it by arguing that the parent (or any ancestor) is the more logical location for a centralized point of synchronization from which to coordinate other tasks.

4.3.3 Deadlock Avoidance Policy

We can now define our runtime deadlock avoidance policy, $\mathcal{P}_{oc}$, which conservatively detects cycles through LCA projection and the concave turn test. We use promise ownership to convert dependences of tasks on promises to dependences of tasks on tasks. By localizing information about arbitrarily large cycles to the vicinity of a single representative vertex using LCA projection, we will be able to implement the policy with a simpler algorithm than by detecting concave turns directly (see the discussion around Lemma 47). The cause of a cycle is understood as invalid synchronization between sibling subtrees and, possibly, their common parent thanks to the modular reasoning afforded by LCA projection: Anything happening a subtree is attributed to the root of that subtree.

Definition 46. The *cycle detection policy*, $\mathcal{P}_{oc}$, extends $\mathcal{P}_o$ with additional state in the form of a multigraph, G^*, initially $(\mathit{Task}, \emptyset)$, and a map, W, from promises to multisets of $\mathit{Task} \times \mathit{Task}$ edges, initially $[\_ \mapsto \emptyset]$. G^* stores the LCA-projected waits-for graph, and W stores which edges in G^* are due to which promises. Recall from Section 3.2 that F is the set of all currently fulfilled promises, and owner maps each promise to its owning task. The rules of the policy are as follows:

1. The **get** p instruction is now implemented as two synthetic instructions in sequence: *validate* p; *block* p.

2. No validation occurs when awaiting a fulfilled promise.

$$\frac{p \in F}{(G^*, W) \Rightarrow (G^*, W)}\ t : \mathit{validate}\ p$$

3. Validation does occur when awaiting an unfulfilled promise. The presence of a concave turn in the updated waits-for graph, as tested by the *concave* predicate, is a runtime error. Let $H = p \notin F$.

$$\frac{e = \mathit{lca}^+(t, \mathsf{owner}(p)) \qquad H \qquad \textbf{assert}\ H \implies \neg \mathit{concave}(G^* \cup e)}{(G^*, W) \Rightarrow (G^* \cup e, W[p \mapsto W(p) \cup e])}\ t : \mathit{validate}\ p$$

4. A wait unblocks when the promise becomes fulfilled.

$$\frac{p \in F}{(G^*, W) \Rightarrow (G^*, W)}\ t : \mathit{block}\ p$$

5. Any edges added in rule 3, are removed once p is set.

$$\frac{}{(G^*, W) \Rightarrow (G^* \setminus W(p), W[p \mapsto \emptyset])}\ t : \textbf{set}\ p$$

We desire a verification procedure which can be safely invoked by tasks concurrently, so it is essential to dwell on some subtle aspects of $\mathcal{P}_{oc}$. A task must not observe a **set** for promise p (that is, neither a task awaiting p *nor* the task fulfilling p can proceed) while any edge created by rule 3 for p still exists in G^*. Were this property violated, concurrent invocations of the verifier could raise a false alarm. For example, suppose task a sets p and then awaits q, owned by b. If b is concurrently awaiting p, the edge (b^*, a^*) must be removed *before* a can return from setting p; otherwise, a false cycle can be created in G^* when a awaits q, adding edge (a^*, b^*).

There are two consequences of the preceding requirement. First, rule 3 must be

atomic; we cannot allow p to be fulfilled after finding that it is unfulfilled but before the waits-for edge is added. Second, **set** p can neither return nor cause any awaiting tasks to unblock until all edge removals in rule 5 are completed.

The interaction of LCA projection with $\mathcal{P}_o$ has an important concrete benefit to a verifier of $\mathcal{P}_{oc}$. The owner of a promise may change during a wait. If we skipped LCA projection and operated on the precise waits-for graph directly, edges would move in response to promise ownership transfers, and this would require additional contentious bookkeeping in the policy's implementation. But since $\mathcal{P}_o$ limits the ownership transfer of promises to the forking of new tasks, it turns out that it is always safe for the LCA projection to use stale ownership information obtained without synchronization. The value of $lca^+(a, b)$ is nearly invariant under replacing b by a descendant of b, and the case where it is not turns out to be unimportant. Thus, if a awaits a promise, p, which it believes to be owned by b, it does not matter if b transfers p to a child task while the wait is in progress. By associating each wait to an invariant representative of the real waits-for edge, there is no need to continually update such edges as promises move down the task tree.

Lemma 47. *If b is neither an ancestor of nor equal to a, then $lca^+(a, b')$ is invariant over all b' that are descendants of or equal to b.*

Proof. By induction over descendants of b. If b' is a child of b, then $lca^+(a, b) = lca^+(a, b')$. Moreover, b' is also neither an ancestor of a nor equal to a. □

The only time one might need to be concerned about stale ownership information is when b is an ancestor of or equal to a. First if $b = a$, then a cannot await a promise concurrently with a change of ownership of the promise away from b because a and b are the same task. Second, if b is an ancestor of a, we necessarily have $b^* = b$ and $b^* \leq_{TJ} a^*$. When ownership is transferred to a new child of b, say b', the correct value of b^* from this point onward is b' by Lemma 47. Yet still we have $b' \leq_{TJ} a^*$;

that is, the relative order of a^* and b^* is invariant, even if b^* is not. Since the waitee, b^*, is the lesser of the two, the concave test only cares about the in-degree of a^*, not the out-degree of b^*, and a^* is not affected by ownership transfer at all, again by Lemma 47. Therefore, it is safe to not update such edges in G^* during ownership transfer.

We have deadlock freedom under $\mathcal{P}_{oc}$ as follows.

Lemma 48. *If an error is not raised under $\mathcal{P}_{oc}$, then G^* never contains a concave turn.*

Proof. G^* initially has no edges. Edges are only added by the *validate* action (rule 3), updating $G^* \Rightarrow G^* \cup e$. If no error is raised, then the hypotheses of rule 3 are always met, and so $\mathit{concave}(G^* \cup e)$ always fails. □

Theorem 49 (Deadlock Freedom)**.** *A program, P, that satisfies $\mathcal{P}_{oc}$ does not get stuck.*

Proof. Suppose for the sake of contradiction that P is stuck. Then every unterminated task is blocked on an unfulfilled promise (the hypothesis of $\mathcal{P}_{oc}$ rule 4 is not met). Since P satisfies $\mathcal{P}_{oc}$, then P also satisfies $\mathcal{P}_o$, by definition. Since $\mathcal{P}_o$ guarantees every promise has an owning task, which may not terminate without fulfilling that promise (Theorem 27), then the waits must form at least one cycle. Since a task can only block on an unfulfilled promise after first validating that promise ($\mathcal{P}_{oc}$ rule 1), then G^* contains the LCA projection of every waits-for edge ($\mathcal{P}_{oc}$ rule 3). Since the waits-for edges contain a cycle, so does G^* (Lemma 42). Therefore, by Lemma 45, G^* contains a concave turn, contradicting Lemma 48. □

The futures primitive imposes a restriction wherein a promise is automatically fulfilled when its associated task completes, which has prior work in the area of deadlock policies, namely Transitive Joins [83] from Chapter 2, and its less permissive

predecessor, Known Joins [22]. The $\mathcal{P}_{oc}$ policy for promises subsumes both of these policies for futures.

Theorem 50. *On programs whose use of promises is restricted to futures, $\mathcal{P}_{oc}$ is at least as permissive as Transitive Joins and Known Joins.*

Proof. The waits permitted by Transitive Joins are a superset of the waits permitted by Known Joins (Theorem 20). Every wait, $a \to b$, that is permitted under Transitive Joins is ordered as $a <_{TJ} b$ (Theorem 17). Let $(a^*, b^*) = lca^+(a, b)$. We also have $a^* <_{TJ} b^*$. Therefore, it is impossible to fail the assertion in $\mathcal{P}_{oc}$ rule 3. □

We also have the following convenient property explaining that it is possible to incorporate the finish construct [17] into the verification paradigm we have presented. A finish block awaits the termination of all tasks spawned transitively within its scope.

Theorem 51. *If a program, P, cannot violate $\mathcal{P}_{oc}$, there is an implementation of the finish construct using promises such that* **finish** $\{P\}$ *cannot violate $\mathcal{P}_{oc}$.*

Proof. Each task should await its immediate children in reverse spawn order. All these waits-for edges are invariant under LCA projection. At no time do these edges form a concave turn together because they are all aligned with $<_{TJ}$. Nor can they form a concave turn with other edges in P, which is shown by induction on a traversal of the task tree: A task is not awaited upon by its parent task until all $<_{TJ}$-preceding tasks have already terminated. □

4.3.4 Implementation of the Projected Waits-for Graph

When an await $a \to b$ arises, per $\mathcal{P}_{oc}$ we must compute $(a^*, b^*) = lca^+(a, b)$ and add (a^*, b^*) as an edge to the multigraph G^*. Once the awaited promise is fulfilled, this edge is removed. However, there is no need to store the graph explicitly. We only need to be able to compute the *concave* predicate over the graph, which tests for concave turns.

Suppose $a^* \leq_{TJ} b^*$. It suffices to record that b^* currently has *some* lesser task awaiting it (or its descendants). Since multiple waits may arise concurrently, however, we must be a little more precise by counting *how many* lesser tasks are awaiting b^* (or its descendants). Conversely, an await $b \to c$ may arise with $c^* \leq_{TJ} b^*$, leading us to record how many lesser tasks b^* (or any of its descendants) is awaiting. These per-vertex counters should be updated when an edge is added or removed in the semantics.

Now the test for concavity is simple: There is a concave turn if b^* is simultaneously 1) being awaited by a non-zero number of lesser tasks and 2) is awaiting a non-zero number of lesser tasks. Of course, there is no need to test the entire graph for this property, since the graph is updated incrementally. We need only to test the greater of the two vertices incident on the edge, e, in $\mathcal{P}_{oc}$ rule 3, and no test is required when remove an edge.

4.3.5 Revisiting the Motivating Example

How does $\mathcal{P}_{oc}$ handle Listing 4.1? First, note the shape of the task tree. The root task spawns a child task, i, for each cell of the array. Task i owns the initial promise in the array cell (line 3) and each promise that it creates (line 8). The task fulfills each promise (line 15) after awaiting two greater (older) and two lesser (younger) siblings (lines 10–13). The programmer's intention is that all these waits are for promises which were fulfilled in the previous round, but because of the data race bug, the waits are sometimes applied to promises in the current round.

Suppose that a deadlock cycle arises for the following cell indices: $0 \to 2 \to 4 \to 3 \to 1 \to 0$. This set of dependences does not change under LCA projection, since all these tasks are siblings. The set exhibits a concave turn, $1 \to 0 \to 2$, because tasks 1 and 2 precede task 0 in the task ordering (they are younger siblings of task 0). Either task 1 will trigger the alarm on line 11 or task 0 will trigger the alarm on

line 13, depending on which tries to modify G^* second, since they both compete for the task 0 node. One attempts to increase the in-degree from lesser nodes while the other attempts to increase the out-degree to lesser nodes.

To explore the full range of capability of $\mathcal{P}_{oc}$, one should also consider what would happen if all the sibling tasks of Listing 4.1 were broken up into multiple subtrees. Then any pair of waits-for edges would fall into one of three cases: 1) they both lie entirely within one such subtree, 2) they both span subtrees, or 3) one lies within a subtree and the other spans subtrees. In the first case, the same reasoning as above explains how $\mathcal{P}_{oc}$ detects a concave turn if it is present. In the second case, the LCA projection lifts the edges to the representative roots of the subtrees, and any concave turn would detected in a cycle at that level. In the third case, the spanning edge is lifted while the internal edge is not, and no concave turn can be detected. It is a consequence of Lemma 42 and Lemma 45 that this third case is not problematic, since, if there is a cycle, even after LCA projection some concave turn must remain, detectable in case 1) or 2).

4.4 Improving Verification with Pseudo-synchronization

The $\mathcal{P}_{oc}$ policy is a conservative check for the existence of cycles. Since it sometimes rejects deadlock-free code, we propose a novel *guard* feature for promises that assists in complying with $\mathcal{P}_{oc}$ without over-synchronizing.

4.4.1 False Positives and Loss of Parallelism

We can ask a few questions to understand if the false positives of $\mathcal{P}_{oc}$ are problematic and to what degree. First, can we always repair a deadlock-free program that violates $\mathcal{P}_{oc}$? Second, can this fix can be performed without losing parallelism to unnecessary synchronization?

4.4.1.1 Repairing Programs with False Alarms

It turns out that false alarms can always be suppressed, at least for structurally deterministic programs, which are those whose synchronization dependences are the same for every run.

Theorem 52. *If P is structurally deterministic and deadlock-free, it is possible to make P comply with $\mathcal{P}_{oc}$ by introducing additional awaits on existing promises.*

Proof. The happens-before (h.b.) relation over the instruction instances in a program is the transitive relation that contains (u, v) if v follows u in program order or if v is a **get** that waits for u, a **set**. This relation is acyclic for P because P is deadlock-free. Any false alarm with respect to $\mathcal{P}_{oc}$ is caused by at least two concurrent **get** instructions on unfulfilled promises; call these waits w_1 and w_2. Let s_1 and s_2 be their corresponding **set** instructions. We must enforce that s_1 happens-before w_2 or s_2 before w_1 to suppress the alarm. If s_1 h.b. s_2, then force s_1 h.b. w_2 by inserting a copy of the w_1 instruction before w_2; likewise, if s_2 h.b. s_1, then insert a copy of w_2 before w_1; if neither case holds, then choose one of the two cases arbitrarily. Suppose for the sake of contradiction that the new s_i h.b. w_j edge creates a new deadlock. Then the original program had w_j h.b. s_i. The original program also had s_j h.b. w_j (by the semantics of promises), and therefore, s_j h.b. s_i by transitivity. Yet we performed the insertion only if s_i h.b. s_j or if s_i and s_j were unordered, so we have a contradiction.

In case adding a new dependence introduces a new false alarm, repeat the process until all false alarms are eliminated. This process will terminate since we do not introduce new promises and will eventually reach a maximally acyclic happens-before graph. □

Listing 4.4 contains an illustration of the idea proposed in Theorem 52. The function repairable can raise a false alarm with line 3 and line 10, in which task

Listing 4.4: Repairing a false alarm.

```
1 void repairable (Promise x, Promise y) {
2   async (y) {  // task a
3     x.get();
4     y.set();
5   }
6
7   async (x) {  // task b
8     async () { // task c
9       // Fix: insert x.get()
10      y.get();
11    }
12    x.set();
13  }
14 }
```

a awaits b while c awaits a. This alarm can be suppressed by the addition of an await on x at line 9, which forces the dependence $a \to b$ to be removed before the dependence $c \to a$ arises. This modification does not over-synchronize the program in this case since the setting of x happens-before the setting of y anyway, due to the program order in task a.

4.4.1.2 Repairing Causes Over-synchronization

Unfortunately, any repair method, not just the one in Theorem 52, has the potential to lose parallelism by introducing unnecessary synchronization. We give a minimal example in Listing 4.5. Tasks a_1 and b_1 are siblings, with children a_2, b_2, respectively. Task a_2 awaits promise x owned by b_1 while b_2 awaits y owned by a_1. After projection to LCAs, these two awaits form an apparent cycle between the representatives, a_1 and b_1. The goal is to order the two awaits by adding additional awaits. The only ways to proceed are to have a_2 await y first, before awaiting x, or conversely to have b_2 await x first. Either action introduces a false dependence not in the original program: either a_2 on a_1 or b_2 on b_1.

A more practically motivated example of the preceding phenomenon is given in

Listing 4.5: A case where repairing a false alarm must introduce unnecessary synchronization.

```
1 void repair_oversynchronizes (Promise x, Promise y) {
2   async (y) {  // task a1
3     async () { // task a2
4       // Fix option 1: insert y.get()
5       x.get();
6     }
7     y.set();
8   }
9
10  async (x) {  // task b1
11    async () { // task b2
12      // Fix option 2: insert x.get()
13      y.get();
14    }
15    x.set();
16  }
17 }
```

Listing 4.6: Exchanging parallelism for safety; recovering parallelism with guards.

```
1 Promise [][] A = new Promise[I][R]; // initialized
2 Promise [] B = new Promise[R];      // initialized
3 for (int i = 0; i < I; i++) {
4   async (A[i]) { // move whole array
5     A[i][0].set(...);
6     for (int r = 0; r < R; r++) {
7
8       // Loss of parallelism                  // Verified pseudo-wait
9       B[r].get();                    ⇒       guard(B[r]) {
10      work(i,r);                               work(i,r);
11                                               }
12
13    }
14  }
15 }
16 async (B) { // move whole array
17   for (int r = 0; r < R; r++) {
18     for (int i = 0; i < I; i++) {
19       A[i][r].get();
20     }
21     B[r].set();
22   }
23 }
24
25 void work (int i, int r) {
26   double v = A[i][r].get();
27   v += A[(i-2)%I][r].get();
28   v += A[(i-1)%I][r].get();
29   v += A[(i+1)%I][r].get();
30   v += A[(i+2)%I][r].get();
31   A[i][r+1].set(v/5);
32 }
```

Listing 4.6. This program is a correct, race-free, deadlock-free variant of Listing 4.1, which is achieved by using unique memory locations for each round. (In practice, one might use a red-black strategy, but this overcomplicates the example.) However, due to the modulo-I access of the array, even its correct deadlock-free synchronization would trigger a false alarm without the addition of line 9 and lines 16–23. The policy would raise an alarm if task $I-1$ awaited task 0 in line 29 while task 0 awaited task 1 in line 29, as this would create a concave turn.

Thanks to the use of the B array, however, this alarm is not raised. Each round completes in its entirety before the next round begins, due to the barrier-like pattern implemented by the get on line 9, the set on line 31, and the task on lines 16–23, which waits for all of $A[\,\cdot\,][r]$ before setting $B[r]$. All of the tasks now proceed in lockstep. Therefore, the awaits of the problematic concave turn occur only *after* those promises are already fulfilled, raising no alarm. But since the exchange of data occurs through promises that already perform the necessary point-to-point synchronization, the await on line 9 is not actually required for the correctness of the program. It would be acceptable for some tasks to proceed through the rounds ahead of others. Therefore, line 9 is only required for the sake of satisfying $\mathcal{P}_{oc}$, at the expense of parallelism.

Our solution to such a dilemma, which we present in the next section, achieves the best of both worlds: combining an easy-to-verify, well-behaved synchronization pattern with a minimally synchronized, but policy-violating pattern. If we could feed the over-synchronized waits mediated by the B array to the verifier, while having the program actually bypass B, then individual workers would be allowed to get ahead of the pack, while the verifier trails behind to check the imaginary well-behaved synchronization that is proceeding only as fast as the slowest worker.

4.4.2 Guards

Since we can modify every deadlock-free program to satisfy $\mathcal{P}_{oc}$, but since this action has a cost of introducing unnecessary synchronization dependences and a loss of parallelism, we propose a novel feature for promise objects. A **guard** block, serves to comply with the policy *as if* by adding synchronization *without* actually restricting the program schedule in any way. For this reason, we say that a guard is a "pseudo-wait."

Definition 53. The language $\mathcal{L}_{pg}$ extends the syntax of $\mathcal{L}_p$ with a new instruction, **guard** (p) $\{P\}$.

A subprogram P can be shielded from the verifier by a guard on promise p. We claim it is safe to execute P *without validating* any of its awaits if we simultaneously perform a *validated* await of p, provided that if p is fulfilled before P terminates, verification resumes on the remainder of P. If P terminates before p is fulfilled, then we abandon the wait on p, removing any record of it from the verifier.

Notice that is not possible to entirely evade the policy through the use of guards. Guards merely lift the reasoning about deadlocks to coarser-grained synchronization by substituting a representative pseudo-wait, which must actually be validated. If it is invalid to await p, entering the guard block raises an alarm. If P deadlocks under the guard, an alarm will be raised once p is fulfilled. If P violates $\mathcal{P}_{oc}$, but only before p is fulfilled, no alarm is raised. So the addition of guards to the language does not break our verification scheme.

Provided that it is safe to await p and that P does not violate $\mathcal{P}_{oc}$ after p is fulfilled, then **guard** (p) $\{P\}$ is semantically equivalent to merely executing P, but with protection against false alarms. Therefore, the programmer can use guards to add "verifier hints" to the program, which serve several purposes:

- False alarms are suppressed as if by the addition of synchronization, and yet no

parallelism is lost; moreover, some validation overhead is avoided.

- The programmer's intention regarding the happens-before relation between certain promise fulfillments is communicated in the code in a coarser, block-structured manner.

- Deadlocks involving violations of this programmer intent are identifiable as such, since an alarm raised from within a guard witnesses the invalid assumptions about the ordering of promises with respect to the guard promise.

In summary, **guard** (p) $\{P\}$ executes P without verification of its awaits unless and until p is fulfilled; meanwhile a verified wait on p is recorded and remains in place until p is fulfilled or P completes. This behavior of guards generalizes $\mathcal{P}_{oc}$ to give our final policy, $\mathcal{P}_{ocg}$. While it is possible to develop a formal semantics of $\mathcal{P}_{ocg}$, it is more enlightening to give a high-level overview and proceed directly to its algorithmic implementation.

Definition 54. The *guard policy* $\mathcal{P}_{ocg}$ extends $\mathcal{P}_{oc}$ as follows. To every task, t, is associated a list, $g(t)$, of guarding promises, initially empty. We introduce two more synthetic instructions, *app* and *rm*.

1. t : *validate* p and t : *block* p behave as in $\mathcal{P}_{oc}$.

2. The **get** p instruction is implemented as *app* p; *block* p; *rm* p. Note that *validate* is no longer invoked directly.

3. The code **guard** (p) $\{P\}$ is implemented as *app* $p; P;$ *rm* p. Note that there is no *block*.

4. The instruction t : *app* p tests if $g(t) \subseteq F$. If so, then it performs t : *validate* p. Regardless, it appends p to $g(t)$.

5. The instruction t : **set** p, in addition to its $\mathcal{P}_{oc}$ behavior, performs the following: For each task t' such that $p \in g(t')$, if every promise preceding p in $g(t')$ is fulfilled, let q be the first unfulfilled promise in $g(t')$, if any, and perform t' : *validate* q.

6. The instruction t : *rm* p removes p from the end of $g(t)$ and removes any edges that may have been added to G^* by t : *validate* p (a subset of $W(p)$), if they have not already been removed.

The $\mathcal{P}_{ocg}$ policy reduces to $\mathcal{P}_{oc}$ on programs without guards.

Theorem 55. *If a program, P, contains no guards, then $\mathcal{P}_{ocg}$ and $\mathcal{P}_{oc}$ are equisatisfiable.*

Proof. Since there are no guards in P, for every task t, values are only appended to $g(t)$ under the $\mathcal{P}_{ocg}$ semantics by t : **get** p and are immediately removed upon completion of the get (rule 2).

The instruction t : *rm* p cannot be invoked until after some task invokes **set** p. But **set** p removes the edges from G^* that were added by any prior *validate* p ($\mathcal{P}_{oc}$ rule 5), and any subsequent *validate* p is a no-op ($\mathcal{P}_{oc}$ rule 2). Therefore, there are no edges remaining in G^* that *rm* p can remove, so the semantics of *rm* p with respect to G^* are a no-op.

Whenever t : *app* p is invoked, $g(t)$ is always empty, so that t : *validate* p is always invoked. Therefore, **get** p has the same semantics under $\mathcal{P}_{ocg}$ as $\mathcal{P}_{oc}$.

Furthermore, $g(t)$ cannot have more than one element, since each *app* p is immediately followed by a *rm* p. The additional semantics of **set** p under $\mathcal{P}_{ocg}$ (rule 5) are not invoked unless there exists a task t' such that $g(t')$ contains at least *two* promises, which is impossible. Therefore **set** p has the same semantics under $\mathcal{P}_{ocg}$ as $\mathcal{P}_{oc}$. □

Even in the presence of guards, which exempts some waits from verification, we still have deadlock freedom. The key to the proof is to see that every blocked task is

participating in *some* validated wait or pseudo-wait (the outer-most unfulfilled guard promise), even if the actual blocking wait has not been validated.

Lemma 56. *During execution of a program under $\mathcal{P}_{ocg}$, for every task t, the first unfulfilled promise, p in $g(t)$, if any, has had an edge added to G^* by t : validate p, and this edge has not been removed.*

Proof. Suppose $g(t)$ has a first unfulfilled promise, p. If p was the first unfulfilled promise of $g(t)$ immediately upon being appended to $g(t)$, then t : *app* p (rule 4) has invoked t : *validate* p. However, if p became the first unfulfilled promise of $g(t)$ later, due to some preceding promise, q, becoming fulfilled, then that **set** q instruction (rule 5) has invoked t : *validate* p. Therefore, the appropriate edge, e, has been added to G^*.

The only two instructions that can remove e from G^* are **set** p ($\mathcal{P}_{oc}$ rule 5) and t : *rm* p ($\mathcal{P}_{ocg}$ rule 6). Since p is unfulfilled, we know **set** p has not been invoked. Since $p \in g(t)$, we know t : *rm* p has not been invoked. □

Theorem 57 (Deadlock Freedom with Guards)**.** *If $P \in \mathcal{L}_{pg}$ satisfies $\mathcal{P}_{ocg}$, then P does not get stuck.*

Proof. For the sake of contradiction, suppose P is stuck. Every non-terminated task t is stuck at a *block* p_t instruction; therefore $g(t)$ contains at least one unfulfilled promise. Let q_t be the first unfulfilled promise in $g(t)$. By Lemma 56, the state of G^* is the same as the LCA-projected waits-for graph for a guard-free program in which every task t is actually blocked on promise q_t. We conclude that G^* contains a cycle and exhibits a concave turn (Lemma 45). But since *concave* is checked every time an edge is added to G^* ($\mathcal{P}_{ocg}$ rule 1, with $\mathcal{P}_{oc}$ rule 3), we have a contradiction. □

Through the use of guard blocks, we can now prove that it is always possible to remove false alarms without losing parallelism.

Theorem 58. *If $P \in \mathcal{L}_{pg}$ is structurally deterministic and deadlock-free, it is possible to make P comply with $\mathcal{P}_{ocg}$ without loss of parallelism by introducing guards.*

Proof. We follow the proof of Theorem 52, but instead of inserting an await, say on promise p, so that it happens-before an existing await, enclose the existing await with a guard on p. Since guarding on p does not in any way restrict the schedule of the program, no parallelism is lost. □

In a real program, better solutions than this proof may exist. Many awaits could be grouped under the same guard, or one guard could be omitted because it is itself guarded by a promise that is always fulfilled after it.

As an example, we return to Listing 4.6, modified this time so that $\mathsf{work}(i, r)$ occurs under a guard on $B[r]$ (line 9), rather than after an await of $B[r]$. As long as $B[r]$ remains unfulfilled (at least until all the workers complete round r), the waits in $\mathsf{work}(i, r)$ are not validated. Yet the program is not made unsafe since the guard on $B[r]$ is validated *as if* it were a real await. A waits-for edge for task i on promise $B[r]$ will be in place either until the guarded code completes or until $B[r]$ is set, whichever comes first. Since the promises awaited in $\mathsf{work}(i, r)$ are definitely fulfilled before $B[r]$ is fulfilled, none of the blocking waits in work ever actually need to be validated. The net effect of this behavior is that the verifier only sees a synchronization pattern that complies with the policy, and yet the program is actually performing $\mathcal{P}_{oc}$-violating synchronization. Guards thus yield a more flexible policy as well as more efficient verification, since fewer waits need to be validated.

4.5 Verifier Algorithm

We now describe the algorithms which run when each of the promise operations is invoked, in order to compute $\mathcal{P}_{ocg}$.

Algorithm 6 Promises and Tasks

```
 1: procedure NEW()
 2:     t ← task_cur
 3:     p ← {owner : t, fulfilled : false,
            lockout : false, wlist : []}
 4:     append p to t.plist
 5:     return p

 6: procedure FORK(proms, f)
 7:     t ← task_cur
 8:     assert p.owner = t forall p ∈ proms
 9:     t' ← {plist : proms, waits : 0, guards : []}
10:     p.owner ← t' forall p ∈ proms
11:     add t' as a new child of t
12:     do asynchronously
13:         task_cur ← t'
14:         f()
15:         assert t'.plist is empty
```

4.5.1 Promises and Tasks

Procedures for the creation of new promises and the transfer of promise ownership at fork time are given in Algorithm 6.

Each promise has fields to store its current owner (initially the current task), whether it has been fulfilled, whether a set is in progress (lockout), and a list of wait records. This list must support the operations CLEAR, ITERATE, and ATOMIC-APPEND, described in Algorithm 7.

Each task has fields to store its currently owned promises, a counter of the number of lesser waiters (positive) or waits on lesser tasks (negative) currently associated to this subtree after LCA projection, and a list of guards currently in effect, which can include a real await as a special case. This list must support the operations APPENDWASEMPTY, COMPAREPOPPEEK, and COMPAREPOPLAST, described in Algorithm 7. Just as in Algorithm 4, a new task adopts a subset of its parent's

Algorithm 7 List Operations

1: **procedure** CLEAR(l) ▷ empties l
2: **procedure** ITERATE(l) ▷ iterates over at least all the elements added by ATOMICAPPEND before iteration begins.
3: **procedure** ATOMICAPPEND(l, e) ▷ appends e to l atomically

4: **procedure** APPENDWASEMPTY(l, e) ▷ atomically appends e to l and returns whether l was initially empty
5: **procedure** COMPAREPOPLAST(l, e) ▷ atomically removes the last element of l if it is e
6: **procedure** COMPAREPOPPEEK(l, e) ▷ atomically removes the head of l if it is e and returns the new head; null otherwise

promises (lines 8–10), and, before termination, checks that it has dispatched every owned promise in some way (line 15), either by fulfilling it or moving it to a new child. Extending the task tree for the purposes of LCA computations occurs in line 11.

Multiple instances of CLEAR and ATOMICAPPEND and at most one instance of ITERATE may be called concurrently. Multiple instances of COMPAREPOPPEEK and at most one instance each of APPENDWASEMPTY and COMPAREPOPLAST may be called concurrently.

4.5.2 Wait Records

The representation of G^* is manipulated by the complementary procedures RECORD and STRIKE in Algorithm 8. A wait, w, includes a status field and the a^*, b^* representatives from lca^+. RECORD atomically performs the test for concave turns and records the existence of wait w if successful (lines 5, 6, and 10), while STRIKE undoes the operation. Crucially, because of the locking (lines 2, 14) and the status field, which changes monotonically from *init* to *recorded* to *struck*, we see that 1) STRIKE is idempotent, 2) STRIKE only strikes a wait that was actually recorded, and 3) if STRIKE is called before RECORD, then the wait will not be recorded at all. On a given wait object, RECORD is called at most once, and STRIKE at most twice, con-

Algorithm 8 Wait Records

```
 1: procedure RECORD(w)
 2:     with lock(w) do
 3:         if w.status ≠ init then return
 4:         w.status ← recorded
 5:         if ORD(w.a*, w.b*) then
 6:             if ¬incIfNonNeg(w.b*.waits) then
 7:                 w.status ← struck
 8:                 assert false
 9:         else
10:             if ¬decIfNonPos(w.a*.waits) then
11:                 w.status ← struck
12:                 assert false

13: procedure STRIKE(w)
14:     with lock(w) do
15:         if w.status = recorded then
16:             if ORD(w.a*, w.b*) then
17:                 dec(w.b*.waits)
18:             else
19:                 inc(w.a*.waits)
20:         w.status ← struck

21: procedure ORD(a*, b*)                ▷ a*, b* are siblings or parent and child
22:     if a* parent of b* then return true
23:     if b* parent of a* then return false
24:     return whether a* is newer than b*
```

currently, so the locks here have very low contention. The counter functions *inc*, *dec*, *incIfNonNeg*, and *decIfNonPos* are atomic. The latter two modify the counter only if the condition on the current sign holds, returning true if the modification was performed.

4.5.3 Wait Validation

The primary procedures for validating waits are in Algorithm 9. The process is nearly the same for awaits and guards except that GUARD omits the blocking wait included in AWAIT (line 3) and must manually ensure that STRIKE has been called before

Algorithm 9 Wait Validation

```
 1: procedure AWAIT(p)
 2:     w ← CHECK(p)
 3:     wait(p.fulfilled_acq)
 4:     COMPAREPOPLAST(task_cur.guards, w)

 5: procedure GUARD(p, f)
 6:     w ← CHECK(p)
 7:     f()
 8:     STRIKE(w)
 9:     COMPAREPOPLAST(task_cur.guards, w)

10: procedure CHECK(p)
11:     a ← task_cur
12:     w ← {task : a, prom : p, status : init, a* : null, b* : null}
13:     if p.fulfilled_acq then return w                          ▷ early out
14:     if APPENDWASEMPTY(a.guards, w) then
15:         VALIDATE(w)
16:         if w.status ≠ recorded then
17:             COMPAREPOPPEEK(a.guards, w)                       ▷ returns null
18:     return w

19: procedure VALIDATE(w)
20:     p ← w.prom
21:     if p.fulfilled then return
22:     assert w.task ≠ w.owner
23:     (a*, b*) ← lca⁺(w.task, w.owner)
24:     ATOMICAPPEND(p.wlist, w)
25:     if p.lockout_acq then
26:         wait(p.fulfilled_acq)                 ▷ short-term wait; set in progress
27:         CLEAR(p.wlist)
28:         return
29:     w.a* ← a*
30:     w.b* ← b*
31:     RECORD(w)
32:     return
```

exiting the guarded block (line 8), since the set may not have occurred yet.

The CHECK procedure constructs a wait object with placeholders for the LCA representatives (line 12). The normal behavior is to then validate the wait, which raises an exception if the wait is invalid at this time (line 15). We append the wait to the current task's guard list (line 14); however, if there is already a guard in effect, we skip validation for now. This guard entry is guaranteed to be removed no later than line 4 or line 9, if it is not already removed by then. If w was not recorded during VALIDATE, we remove it from the guard list, as the promise is already fulfilled (lines 16, 17); it is possible that w is found not to be recorded for another reason (it *has been* recorded and is now struck), in which case another party may concurrently try to remove it from the guard list, hence the atomic operation COMPAREPOPPEEK.

VALIDATE checks $\mathcal{P}_{oc}$, regardless of the guards that are in effect; it is invoked proactively by CHECK when there are no guards, and retroactively by SET when a guard is removed. Line 21 is not an early out; it is required for correctness if a task awaits a promise which it owns after having fulfilled it. We raise an exception if a task awaits its own promise prior to fulfilling it (line 22). We then compute the LCA representatives for w (line 23). The wait object is next added to the list of waits on the promise (line 24) so that SET can strike the wait later. The normal behavior is that the set has not yet begun, so we record the wait in lines 29–31. But if we find that the set is in progress, we should not record w at all and instead wait for SET complete its work of striking waits (lines 25–28).

4.5.4 Setting

Algorithm 10 gives the SET procedure. The promise, p, is removed from the list of promises owned by its task (line 4). We announce that the set has begun so that no new waits will be added to $p.\mathsf{wlist}$ (line 5). Each wait on the promise must now be struck (lines 7, 8). Additionally, if we are striking the outer-most guard of a task,

Algorithm 10 Setting

```
 1: procedure SET(p)
 2:     t ← task_cur
 3:     assert p.owner = t
 4:     remove p from t.plist
 5:     p.lockout_seq ← true
 6:     R ← ∅                                              ▷ waits to re-check
 7:     for w ∈ p.wlist do                                   ▷ via ITERATE
 8:         STRIKE(w)
 9:         w ← COMPAREPOPPEEK(w.task.guards, w)
10:         if w ≠ null then
11:             R ← R ∪ {w}
12:     p.fulfilled_rel ← true
13:     CLEAR(p.wlist)
14:     for w ∈ R do
15:         g ← w.task.guards
16:         while w ≠ null do
17:             VALIDATE(w)
18:             if w.status = recorded then break
19:             w ← COMPAREPOPPEEK(g, w)
```

then we remove that guard and mark the next one for rechecking (lines 9–11). The critical work for SET is now completed, so we can release any waiters (line 12). Now we recheck the waits no longer protected by a guard (line 14). We iterate down the guard list that the exposed wait belongs to, trying to validate each entry until we confirm that the top-most guard is now a recorded (hence validated) wait (lines 15–19). Every wait we encounter for a promise that has already been fulfilled is removed (line 19). The interaction between line 19 and CHECK's line 14 ensures that every wait either 1) is added to an empty guard list and is, therefore, validated originally (CHECK line 15), or 2) is added to a non-empty guard list and is still guarded by the recorded wait w on line 18 of SET or discovered to be unguarded on line 19 and validated retroactively at line 17. It is possible that the recheck phase of one SET could concurrently try to visit a wait being struck by another SET, so multiple parties will try to recheck the same guard list; the atomicity of COMPAREPOPPEEK

between line 9 and line 19 ensures that exactly one party will succeed. The number of concurrent calls to the list methods on a given guard list is bounded by one more than the length of the list, which is usually very small.

4.6 Evaluation

We compare our implementation of dynamic deadlock detection using $\mathcal{P}_{ocg}$ against an unverified baseline and the precise cycle detector from Section 3.4 that traverses the waits-for path out of each awaiting task. Both $\mathcal{P}_{ocg}$ and the precise detector share the same infrastructure for tracking promise ownership. This implementation makes a different trade-off in computation time versus memory usage when maintaining the lists of owned promises, compared to the implementation in Chapter 3. Therefore, the results presented here are not comparable with Section 3.5.

Because we have introduced the guard block primitive, which is a no-op to the actual program semantics and only meaningful to our proposed verifier as a mechanism to reduce false alarms, we note that the precise cycle detector simply treats **guard** (x) $\{P\}$ as just P.

Our benchmarks consist of eight parallel Java programs using promises, some drawn from prior work and others written for this work.

1. Conway simulates a 2D cellular automaton on a 3000×3000 grid with 100 workers. The tasks synchronize directly with their neighbors, guarded in a manner similar to Listing 4.6.

2. Heat simulates diffusion using the 1D heat equation. 50 tasks operate on chunks of 40,000 cells for 5000 iterations. The synchronization pattern is similar to Listing 4.6, using guards to satisfy $\mathcal{P}_{ocg}$ instead of over-synchronizing with a barrier.

3. QSort sorts 1 million elements using an async-finish divide-and-conquer recur-

sion that is parallelized until the chunks have fewer than 20 elements, so the tree has depth 16. Finish is implemented using promises.

4. Sieve counts the primes below 100,000 with a pipeline of tasks, each filtering out the multiples of an earlier prime.

5. SmithWaterman (adapted from HClib [41]; also used in prior work [83, 22]) aligns DNA sequences having 18,0000-20,000 characters each. Each task operates on a 25×25 tile.

6. Strassen (similar programs may be found in the Cilk, BOTS, and KASTORS suites [31, 27, 81]) performs 256×256 matrix multiplication on sparse matrices with about 8192 entries. The divide-and-conquer recursion issues asynchronous tasks for both addition and multiplication phases, up to depth 5, and individually awaits tasks when each result is required.

7. StreamCluster (adapted from PARSEC [8]) computes an online k-means clustering of 102,400 128-dimensional points using 8 worker tasks. The original OpenMP implementation uses barriers, which we implemented using promises.

8. StreamCluster2 is our variation on StreamCluster, which over-synchronizes with barriers. It is only necessary for task 0 to await tasks 1–7. We relaxed the synchronization to show that the precision of promises can be helpful.

All benchmarks were run with Java 8 on a Linux machine with a 16-core AMD Opteron processor. Asynchronous tasks were scheduled by a thread pool with no limit on the number of threads. We measured both execution time and memory usage, taking averages over thirty runs within the same Java VM, after discarding five warm-up runs to mitigate JIT compilation noise, which is a standard technique [34].

Table 4.1 gives the unverified baseline measurements for each program and the overhead factors introduced by each of the verifiers. We give the geometric mean of

Table 4.1: Execution time and memory overheads for verification of promises using cycle detection and an LCA-based policy.

Benchmark	Time (s)/ Memory (GB)	Overheads cycle detection	Overheads $\mathcal{P}_{ocg}$
Conway	4.7259	0.89×	0.88×
	4.6944	1.03×	1.04×
Heat	5.3612	1.00×	1.01×
	0.0931	1.35×	2.31×
QSort	18.6821	1.02×	0.70×
	0.2168	1.10×	0.88×
Sieve	2.5281	1.07×	1.73×
	2.4879	1.30×	1.64×
SmithWaterman	56.8619	1.00×	0.99×
	2.3079	0.99×	1.22×
Strassen	1.3274	0.96×	0.94×
	0.8128	1.58×	1.51×
StreamCluster	34.6145	1.03×	1.10×
	0.1331	1.36×	3.08×
StreamCluster2	24.5509	0.99×	1.07×
	0.1069	1.23×	2.07×
Geometric Mean Overhead	Time	1.00×	1.02×
	Memory	1.20×	1.59×

overheads across all benchmarks.

Overall, both the precise cycle detector and $\mathcal{P}_{ocg}$ introduce very little execution time overhead. On Conway, both verifiers actually reduce the time by more than 10%, and on QSort, $\mathcal{P}_{ocg}$ reduces it by 30%. These reductions may seem counterintuitive, but the additional CPU time required when awaiting an unfulfilled promise and, in the case of $\mathcal{P}_{ocg}$, when fulfilling a promise with many waiters, can significantly perturb the scheduling of tasks onto threads and affect the number of threads spawned by the pool. The only benchmark for which $\mathcal{P}_{ocg}$ exceeded 1.1× execution time is Sieve, with a factor of 1.73×. Compared to all the other benchmarks, Sieve tasks are synchronization-heavy and perform almost no actual computation.

Memory overheads vary across the benchmarks for both verifiers. $\mathcal{P}_{ocg}$ introduces the largest memory overheads on StreamCluster (3.08×) and Heat (2.31×). We note that nondeterminism in scheduling the program may provoke variability in garbage collection, and it would be interesting to perform similar tests in a language with explicit memory management.

4.7 Conclusion

We defined a deadlock-freedom policy for programs using promises for synchronization. The policy relies on promise ownership, task tree LCAs, and an ordering over tasks induced by the task tree. Complying with our policy makes parallel code easier to reason about in a modular way by ensuring that synchronization between subtrees of tasks is coordinated. The effect of our policy is invariant under passing ownership of a promise to a new child task so that a verification algorithm can be implemented without contentious bookkeeping.

We introduced a novel pseudo-synchronization primitive, the guard block, which interacts with our policy as if by awaiting a promise without actually blocking. Code inside such a block can safely proceed without validation until the guarding promise

is fulfilled. Deadlock freedom is preserved since the guarding wait itself is validated. This primitive allows programmers to write complex, fine-grained synchronization patterns that would otherwise violate our policy, while still providing a structured style.

We provided algorithms for dynamic verification of the policy that are correct with non-contentious synchronization on global state. An implementation of our verification algorithms introduces negligible execution time overhead on average with respect to a baseline and 1.31× geometric mean memory overhead compared to 1.29× for a precise cycle detector.

CHAPTER 5
SUUBPHASES: IMPROVING THE DEADLOCK-FREE SEMANTICS OF PHASERS

5.1 Introduction

The *phaser* [74] is a significantly more complex synchronization mechanism than the futures and promises that were addressed in Chapters 2–4. One phaser shared by a group of tasks can act like a cyclic barrier, requiring every task to arrive at each round before allowing any to proceed. More generally, however, each task can register in one of several modes indicating whether that task is one of those responsible for signaling the phaser and, separately, whether that task must wait to be released by the phaser. To guarantee sufficient conditions for deadlock freedom, phasers can be subjected to a restricted usage pattern that is deadlock-free by construction [74].

However, deadlock freedom for phasers comes at a cost to program modularity. One of the restrictions imposed on the interface of phasers is that it is not legal to await specific phasers; doing so could introduce a deadlock if the programmer does not use a consistent phaser ordering discipline when interleaving signals and waits. Instead, a task must await *every* phaser to which it is currently registered with the wait capability, and, moreover, this action must be preceded by that task's signaling of all phasers to which it is registered with the signal capability. The bundled series of signals and awaits is invoked by a single call to a function called **next**. This forced interaction with all registered phasers can cause code to interact needlessly with irrelevant phasers. The entangling of synchronization *between* two sections of code with

the synchronization *within* each section leads to over-synchronization and an anti-modular programming style. We explore examples of this undesirable consequence of deadlock-freedom constraints and propose a solution that works by 1) organizing phasers and their operation into levels, and 2) restricting the behavior of **next** based on these levels to recover modularity.

It is worth noting that implementations of the phaser can vary across languages. Unlike the implementations in X10 [74] and in Habanero Java [15], which behave as described above, the Java phaser does not use capability registration and does not enforce or guarantee deadlock freedom by restricting waits [69]. In this work, we target the richer Habanero phaser that has capability registration and a deadlock-freedom guarantee [74].

5.1.1 Phasers

Computations amenable to phaser synchronization are those in which parallel tasks operate on shared memory and require multiple blocking dependences to ensure data is available or safe to overwrite. A long computation can be divided into multiple *phases* with inter-task dependences occurring at the phase boundaries. Often, such dependences must be repeatable because they occur inside a loop. A single phaser object can be used to order the phases across many tasks.

As an introductory example to the kinds of computations for which phasers are useful, Listing 5.1 shows three asynchronous tasks iterating in parallel. A series of values is produced by each task, with the ith element of array b depending on the $(i-1)$st elements of arrays a and c (line 12). To establish the dependences, the three tasks are registered to a common phaser, ph. The first and third tasks are producers of the dependences and so are registered in signal mode to ph (lines 3, 15). Since the second task is a consumer, it is registered in wait mode (line 9). The behavior of this phaser is that the ith call to `next()` by the second task blocks until after *both* ith calls

Listing 5.1: Typical example of phaser behavior.

```
1 Phaser ph;
2 double [] a, b, c;
3 async (SIG(ph)) {
4   for (int i = 0; i < I; i++) {
5     a[i] = workA(a[i-1]);
6     next();
7   }
8 }
9 async (WAIT(ph)) {
10  for (int i = 0; i < I; i++) {
11    next();
12    b[i] = workB(a[i-1], c[i-1]);
13  }
14 }
15 async (SIG(ph)) {
16  for (int i = 0; i < I; i++) {
17    c[i] = workC(c[i-1]);
18    next();
19  }
20 }
```

to next() of the first and third tasks. There are no other blocking dependences among the other calls to next(). This behavior is established by the choice of registration modes. Internally, the next function advances the phase number of each task with respect to the phaser. In the first phase of computation, the first instances of lines 5 and 17 execute in parallel. In the second and further phases, line 12 executes in parallel with the subsequent instances of lines 5 and 17. In the final phase, the final instance of line 12 executes. However, since the first and third tasks are not waiters, they are permitted to execute their phases 1) not in step with one another and 2) arbitrarily far ahead of the second task's phases. This is a distinguishing feature of phasers that generalizes cyclic barriers, in which every registered task is both a signaler and a waiter. In more complex programs, a collection of many tasks can communicate via a phaser by signaling, waiting, or both, and there may be multiple phasers coordinating different subsets of tasks. For example, when we come to Listing 5.4, we will see a program that applies an iterative averaging algorithm in

which worker tasks communicate with each another using one phasers and with the root task using a separate phaser.

Some accounting occurs in the implementation of the phaser to determine when a given waiter's next may be released depending the number of next calls executed by each signaler. We will now walk through some increasingly sophisticated variants of the phaser, which govern this accounting. The introductory example just given uses the Habanero phaser (Section 5.1.1.3).

5.1.1.1 Java Phasers

At the lowest level, a phaser is a synchronization object with a parameter k, indicating the number of *parties*, and **signal** and **wait**(n) methods having the following semantics:

1. The phaser's internal *phase number* is a counter that is incremented after every k **signal** operations on the phaser.

2. When a task initiates a **wait**(n) operation on the phaser, the task blocks until the phase number reaches the target value, n.

Unlike a cyclic barrier, where tasks always interact by both signaling and waiting, a task may interact with a phaser by signaling or waiting or both. A simple improvement to the semantics adds register and deregister operations, which allows the number of parties to change dynamically. The preceding low-level phaser semantics, together with such counter-based registration, describes the java.util.concurrent.Phaser class in Java [69].

5.1.1.2 Task-aware Phasers

Despite use of the word "registration," the semantics so far is task-agnostic. The parties do not yet correspond to a specific set of k tasks. The semantics can be encapsulated by a higher-level *capabilities* interface, allowing for dynamic registration

of tasks in various modes. The association of a phaser with the specific set of its registered tasks is accompanied by a richer semantics for **signal** and **wait** [74]:

1. In order to signal a phaser, a task must be registered with the SIG capability. In order to await a phaser, a task must be registered with the WAIT capability. A task may be registered with both capabilities.

2. A task is registered with both SIG and WAIT capabilities for every phaser it creates.

3. Upon creation of a new task, the task may inherit any subset of its parent's capabilities. This subset is specified as an annotation on the task creation primitive.

4. A task may drop any capability at any time.

5. When a task terminates, it implicitly drops all capabilities.

6. Every task records the last phase it *observed* from each phaser, defined below in 8 and 9.

7. The phaser's phase number is set to n once each of its SIG tasks has observed phase $n-1$ and has subsequently invoked **signal** on the phaser.

8. The parameter for the **wait** operation becomes implicit: Its value is n if $n-1$ is the last phase the task observed from this phaser. After the wait, the task's last observed phase for this phaser becomes n.

9. For a task without the WAIT capability, the task's last observed phase for a given phaser is incremented every time the task invokes **signal** on the phaser.

Under this semantics, it now matters not only how many tasks are registered to each phaser, but *which* tasks. Note also that when a task registered in SIGWAIT mode

issues multiple **signal** operations with no intervening **wait**, only the first **signal** is effectual since the last observed phase is not incremented.

5.1.1.3 Habanero Phasers

The final aspect of phaser semantics is a restricted interface that enforces their deadlock-free use by coordinating a given task's interactions with all of its phasers as a group. This represents the most sophisticated definition of phaser, which may be found in Habanero Java and X10 [74].

1. The **wait** operation may not be invoked directly by the user (though **signal** may be).

2. A **next** operation is introduced, which is global in the sense that it does not refer to a specific phaser. It has the effect of signaling every one of the task's SIG phasers and then awaiting every one of its WAIT phasers.

3. A task effectively signals each SIG phaser only once for every phase. This means that after a manual **signal** of a phaser, subsequent signals (including the implicit signal within **next**) have no effect until after the subsequent **next** operation.

4. (A further rule integrates phasers with **finish** blocks in a deadlock-free manner; however we will not discuss it until Section 5.3.3.)

The capability registration and this restricted interface effectively form a deadlock freedom policy for phasers. By replacing the **wait** operation with **next**, we ensure that on every round, every task fulfills all its outstanding obligations to signal its SIG phasers *prior* to awaiting any phasers. Because of the syntactic restrictions, very little of the policy remains to be verified dynamically. The runtime needs only to check the capabilities.

5.1.2 Anti-modularity of a Global 'Next' Operation

Deadlock freedom for phasers is obtained primarily through the semantics of **next**, which is a coarse constraint on the manner in which phasers may be used. For safety, a task may not manually invoke **wait**. The consequence is that if a task is registered to multiple phasers (in any modes), then the synchronization logic becomes entangled through *all* of these phasers. The signals issued by the task and the phase numbers observed by the task must all advance in lockstep with one another, even if there is no need for a logical relationship between some of the phasers. Moreover, this unnecessary connection between phasers can be contagious as it necessarily touches every other task registered to any of those phasers.

Listing 5.2 presents one of the simplest examples of the anti-modularity of **next**. Suppose the programmer wants to synchronize tasks a and b using phaser p, and tasks b and c using phaser q, but that the program logic requires two synchronizations between b and c for every one synchronization between b and a. Ideally, then, task b would advance two phases of q for every one phase of p. However, since b may not elect to await q without also awaiting p or elect to signal q further than one phase ahead of p, the phases of p and q must advance essentially in lockstep. This coupling forces the programmer to engineer task a so that it ignores, say, the odd-numbered phases of p, resulting in the double **next** (lines 26–27). But this constraint on program design was introduced by the need for task b to synchronize with task c. This factor ought to have nothing to do with task a, and yet task a must be designed to accommodate it.

Under the deadlock-free usage policy for phasers, it is not possible to engineer this synchronization pattern without 1) over-synchronizing tasks a and b and 2) requiring task a to be aware of the internal design of task b insofar as b spawns and then synchronizes with c, which has nothing to do with a. In designing task b, the programmer could have called workC1 and workC2 directly instead of spawning c for

Listing 5.2: Basic phaser pattern exhibiting anti-modularity.

```
1 // Task a
2 Phaser p = new Phaser();
3 async (SIGWAIT(p)) {
4   // Task b
5
6   Phaser q = new Phaser();
7
8   async (SIGWAIT(q)) {
9     // Task c
10    while * {
11      workC1();
12      next(); // q
13      workC2();
14      next(); // q
15    }
16  }
17  while * {
18    workB1();
19    next(); // p (undesired), q
20    workB2();
21    next(); // p, q
22  }
23 }
24 while * {
25   workA();
26   next(); // p (undesired)
27   next(); // p
28 }
```

Listing 5.3: Augmenting Listing 5.2 with subphases recovers modularity.

```
1 // Task a
2 Phaser p = new Phaser();
3 async (SIGWAIT(p)) {
4   // Task b
5   Phaser q = subphase {
6     new Phaser();
7   };
8   async (SIGWAIT(q)) {
9     // Task c
10    while * {
11      workC1();
12      subphase { next(); } // q
13      workC2();
14      next(); // q
15    }
16  }
17  while * {
18    workB1();
19    subphase { next(); } // q
20    workB2();
21    next(); // p, q
22  }
23 }
24 while * {
25   workA();
26
27   next(); // p
28 }
```

this work (lines 8–16). That the design of a is dependent in this way on the design of b for reasons beyond the scope of meaningful communication between a and b is evidence that the present deadlock-freedom policy is worth improving.

5.1.3 Contribution

We introduce a novel programming construct called the **subphase** block which allows patterns like Listing 5.2 to be written without over-synchronization or anti-modular design, and yet preserves deadlock freedom for phasers. The use of subphases can recover performance that is lost by using deadlock-free Habanero phasers instead of the lower-level task-aware phasers. In some cases, subphases enable synchronization patterns to be expressed in ways that are more performant than Java phasers, indicating that this novel construct fundamentally improves the expressivity of phaser synchronization.

A subphase block allows some phasers to advance for many phases within a single phase of other phasers. In the first example (Listing 5.2), we desired, but could not achieve, a 2:1 ratio between the phases of q and p. However, in general, there will be no need to fix or even determine the desired ratio in order to make use of subphases. By simply wrapping certain regions of code in a subphase block, the programmer can refine the manner in which the **next** operation interacts with the phasers, synchronizing on certain subsets of a task's registered phasers. Subphase blocks may also be nested, giving rise to composability.

Refer to Listing 5.3 for a subphase-based solution to the anti-modularity of **next** seen in Listing 5.2. Two changes have been made: the introduction of subphase block delimiters in key locations (lines 6, 12, and 19) and the removal of the auxiliary **next** operation (line 26).

The first effect of subphase blocks is to assign phasers to *levels*. Phaser p, created outside any subphase block, is a level 0 phaser, while phaser q, created within a

subphase block (line 6), is a level 1 phaser.

The second effect of subphase blocks is that tasks execute phaser operations at various levels. Here, tasks b and c repeatedly enter and exit a subphase block (lines 12, 19), alternating between levels 0 and 1. Task a resides entirely outside of all subphase blocks, and so is always at level 0.

The semantics one might hope for is that a **next** command issued at level i should operate only on phasers assigned to level i. However, this simple solution is not workable because we would be unable to answer questions about which tasks a phaser should expect signals from at any given time, which varies depending on whether tasks elect to enter subphase blocks or not. (See Section 5.3.5.)

What actually occurs in our approach is that a **next** at level i affects phasers at levels *at least* i. Thus, while line 21 affects both p and q, we exempt line 19 from affecting p by enclosing it in a subphase block. The corresponding **next** instructions in task c are placed into matching subphase levels (0 for line 14, and 1 for line 12). Line 27 does not affect q because task a is not registered to q at that point.

In the remainder of this chapter, we will demonstrate the effective use of subphases on further example programs, showing how they lead to a reduction in unnecessary synchronization (Section 5.2). We will define precise semantics for the operation of **subphase** blocks in conjunction with **next** (Section 5.3), and present algorithms for implementing the behavior (Section 5.5). We will prove that the existing deadlock-freedom usage policy for phasers is still applicable and correct (Section 5.4). Finally, we evaluate an implementation of our approach on benchmark programs that demonstrates how subphases can eliminate enough wasteful synchronization to improve program performance (Section 5.6). Each benchmark is rendered in three styles: 1) using direct **signal** and **wait** instructions for a baseline of the ideal synchronization (Java phasers), 2) using global **next** to guarantee deadlock-freedom at the cost of over-synchronization (Habanero phasers), 3) using global **next** with **subphase** blocks to

reduce synchronization (our approach). In one case, the use of subphases is powerful enough to remove even some unnecessary synchronization that is present in the supposedly ideal baseline.

5.2 Subphase Overview

Following a brief sketch of how the **subphase** block should affect the global **next** instruction, we will explore two more complex examples of programs that benefit from our approach.

5.2.1 Informal Behavior

We introduced the concept of a **subphase** block by explaining that phasers have an allocation level and tasks have a dynamic execution level at which they perform phaser operations. Roughly, a level i **next** instruction signals and awaits phasers at levels *at least* i. For a **next** operation at level i and a phaser, ph, at level j, we have the following two informal rules:

1. If $j < i$ then do not interact with ph.

2. If $j \geq i$ then effectively "fast forward" ph (both in the sense of repeatedly signaling and repeatedly awaiting it) past the **next** instructions issued by other tasks at levels strictly greater than i, returning once all relevant parties have issued **next** at a level less than or equal to i.

The key feature of the forthcoming semantics that allows us to achieve rule 2 is that we will replace the integer phase number with something akin to a fractional phase number. For now, it suffices to say that the number can take on values between integers. For $i = 0$, **next** increments phase numbers by one, as usual. As i increases, **next** uses smaller and smaller increments. Moreover, **next** always *rounds up* the phase

Listing 5.4: Using low-level phaser operations to test for task termination. [19]

```
1 double [] a = new double[I+1];
2 Phaser b = new Phaser();
3 Phaser c = new Phaser();
4 for (int i = 1; i <= I; i++) {
5   async (SIGWAIT(c), SIG(b)) {
6     for (int j = 1; j <= J; j++) {
7       double l = a[i-1];
8       double r = a[i+1];
9       c.signal();
10      c.wait();
11      a[i] = (l + r) / 2;
12      c.signal();
13      c.wait();
14    }
15    c.drop();
16    b.drop();
17  }
18 }
19 c.drop(SIGWAIT);
20 b.drop(SIG);
21 b.wait();
```

numbers it encounters to a degree that matches the increment size; this is the essence of the "fast forwarding" of rule 2.

It will be necessary to pin down what we mean by "small" phase number increments and "rounding up." Unfortunately, it is insufficient to use real number arithmetic for this purpose. We will instead have to employ a more esoteric number system. Fortunately, the algorithmic implementation is not difficult or esoteric and is a natural generalization of existing phaser semantics. And although we have described fast forwarding in terms of repeated signaling and waiting, the implementation need not take repeated actions to achieve the effect.

5.2.2 Phasers for Task Termination

As an example of the restrictive nature of the phaser's traditional deadlock-free use, we can look to the iterative averaging program found in work on the Armus deadlock detector [19], which dynamically identifies phaser deadlocks. An iterative averaging

Listing 5.5: The deadlock-freedom constraints interfere with the operation of multiple unrelated phasers.

```
1 double [] a = new double[I+1];
2 Phaser b = new Phaser();
3 Phaser c = new Phaser();
4 for (int i = 1; i <= I; i++) {
5   async (SIGWAIT(c), SIG(b)) {
6     for (int j = 1; j <= J; j++) {
7       double l = a[i-1];
8       double r = a[i+1];
9       next();
10      a[i] = (l + r) / 2;
11      next();
12    }
13    // Implicit drop of b and c
14  }
15 }
16 c.drop(SIGWAIT);
17 b.drop(SIG);
18 for (int j = 1; j <= J; j++) {
19   next();
20   next();
21 }
```

program is presented in that work as one which cannot be written in a manner that complies with the deadlock-freedom policy.

To illustrate the generality of phasers, the program uses one phaser to synchronize iterations among worker tasks and a second phaser to allow the main task to await the termination of the workers. (Typically, one would use futures or a finish block to await task termination, but phasers subsume this use case.) Using the low-level phaser-specific **signal** and **wait** operations, this program could be rendered as in Listing 5.4. Notice that the worker tasks choose to advance phaser c twice in every iteration (lines 9–10 and lines 12–13), but they release phaser b only once, upon task termination (line 16). The root task can await the termination of all workers by awaiting b (line 21).

While certainly deadlock-free, this pattern cannot be encoded in a way that satisfies the phaser deadlock-freedom policy without the introduction of a large amount of

Listing 5.6: Subphase blocks remove wasteful synchronization.

```
1 double [] a = new double[I+1];
2 Phaser b = new Phaser();
3 subphase {
4   Phaser c = new Phaser();
5   for (int i = 1; i <= I; i++) {
6     async (SIGWAIT(c), SIG(b)) {
7       for (int j = 1; j <= J; j++) {
8         double l = a[i-1];
9         double r = a[i+1];
10        next();
11        a[i] = (l + r) / 2;
12        next();
13      }
14      // Implicit drop of b and c
15    }
16  }
17  c.drop(SIGWAIT);
18 }
19 b.drop(SIG);
20 next();
```

unnecessary synchronization and unproductive work. That policy requires the worker tasks to advance *both* phasers c and b upon every iteration. This is shown in Listing 5.5 with the global **next** operations on line 9 and line 11. While not particularly problematic or disruptive to the workers, a secondary consequence of this change to the code is that now the root task must iterate as well, pumping phaser b not once but $2J$ times and performing no meaningful work in the process (lines 18–21).

We can instead employ our novel **subphase** block as seen in Listing 5.6. Only rule 1 of Section 5.2.1 is in play in this program: **next** does not interact with phasers created in outer subphase levels. Since c is created within the subphase block (line 4), while b is created outside it (line 1), the **next** operations within the block (lines 10, 12) will interact with c, but not with b. In this way, we have modularized the worker tasks' internal synchronization separately from the synchronization with the root task. Upon termination, each worker task implicitly drops all its capabilities, which releases b just once (line 14). The root task then needs only to await b once (line 20).

Fixing the synchronization of this short iterative averaging program can alternatively be achieved through the use of a finish block to await task termination [17]. We will discuss finish blocks in more depth in Section 5.3 and Section 5.4. However, it is merely a coincidence of the simplicity of the example program that a finish-based solution is possible. If the root task needed to await some intermediate condition of the workers, rather than their termination, a finish block would not be of service. Such a more complex scenario follows.

5.2.3 Subphase Blocks of Unspecified Duration

A feature of the preceding example that we can generalize to show how rule 2 of Section 5.2.1 is useful is to remove the fixed iteration count, J. If it is not known up front how many phases a worker will require, then an additional mechanism is required in a program like Listing 5.5 to detect when the root task should stop pumping its phasers. The programmer must ordinarily encode this mechanism by hand. However, subphase blocks do not require any such additional mechanism. The completion of a worker's subphase block can be signaled by performing **next** immediately after the block so that another task can await the block's completion. This synchronization pattern does not depend on knowing the number of phases that elapse *within* the subphase block and works in a more general setting than the finish construct.

We illustrate such synchronization on the completion of a subphase block of unknown duration in our next example, Listing 5.7, which is a QR iteration algorithm for finding matrix eigenvalues. The basic QR iteration algorithm [29] is coordinated by the root task (lines 28–37) and involves conjugating an initial matrix, A, by a series of orthogonal matrices (lines 17–19 together with lines 31–32) until it converges to a diagonal matrix. Each orthogonal matrix, Q, is computed by decomposing the current value of A as QR, where R is upper triangular.

The underlying algorithm for performing the QR decompositions is a parallel

Listing 5.7: QR Iteration, with subphases.

```
1  Matrix A = new Matrix(N,N);
2  Phaser iter = new Phaser();
3  Phaser qr = subphase { new Phaser(); };
4  Rotation [] Q = new Rotation[N*(N-1)/2];
5  AtomicInteger i = new AtomicInteger(0);
6  shared boolean converged = false;
7  // Launch workers
8  for (Partition part : partition(A)) {
9    async (SIGWAIT(iter), SIGWAIT(qr)) {
10     while (!converged) {
11       subphase {
12         // QR by Givens Rotations
13         int stage = 0;
14         for (Entry e : part) {
15           while (!elligibleToEliminate(e, stage++))
16             next(); // qr
17           Rotation r = eliminatingRotation(e);
18           A.leftApply(r);
19           Q[i.getAndIncrement()] = r;
20         }
21       }
22       next(); // iter, qr
23       next(); // iter, qr
24     }
25   }
26 }
27 qr.drop(SIGWAIT);
28 // QR iteration
29 while (!converged) {
30   next(); // iter
31   for (Rotation r : Q)
32     A.rightApply(r);
33   if (isConverged(A))
34     converged = true;
35   i.set(0);
36   next(); // iter
37 }
```

Givens Rotation method [70], performed by a number of worker tasks (lines 8–26). In this instance, Q is implicitly stored as a sequence of rotations, which are successively applied to A, first on the left (line 18) to form R, and then later on the right (line 32) to complete the conjugation. Each worker is responsible for eliminating a partition of the sub-diagonal entries in A. The noteworthy aspect of this approach is that there is a non-trivial dependence graph dictating when entries become eligible for elimination so that concurrently applied rotations do not interfere. We have abstracted away the dependence computations in line 15. All the workers synchronize on the qr phaser in order to traverse the dependence graph one stage at a time. Once an entry is ready to be eliminated, the appropriate rotation (line 17) is applied in-place to A (line 18) and appended to Q (line 19). If multiple entries are simultaneously eligible for elimination, then it does not matter in what order those rotations are added to Q.

Neither the root task nor the workers themselves are required to know how many iterations each worker will use to eliminate its entries. As the workers proceed through the matrix, the available parallelism varies from 1 to $N/2$ and back to 1 again. Therefore, some workers necessarily finish early no matter how the partitioning of entries is performed. When a worker completes its partition, it exits the subphase block and issues **next** (line 22) so that other workers remaining in the subphase block will no longer wait for it until they, too, exit the subphase block and issue **next**. The root task, being outside the subphase block, waits only on the iter phaser (line 30) which cannot advance until all workers have exited the subphase block and issued **next**.

The key benefits of subphase blocks in this program are 1) that a number of qr phases can elapse within a single iter phase, yet this number is not known and moreover varies from worker to worker, and 2) iter waiters (both the root and the workers) can await the completion of the entire group of qr phases without wastefully pumping qr.

The careful reader may suspect that the two phasers in Listing 5.7 can be conflated into a single phaser. This is correct. The **next** operation in line 22 awaits iter and additionally awaits the completion of the entire group of qr phases belonging to the subphase block (lines 11–21). However, these two wait conditions are always fulfilled together in this program. Upon eliminating line 2 and line 27 and removing the iter capability from the **async** annotation in line 9, the program retains the same effective synchronization semantics. However, due to the underlying implementation of phasers, the single-phaser version of this program may actually incur more overhead than the two-phaser version shown. In the single-phaser version, while the **next** operation in line 30 is blocked, the root task is repeatedly awakened upon each instance of **next** in line 16 to re-check the current phase number. In the two-phaser version, this behavior is not seen because these two **next** operations involve disjoint sets of phasers.

5.3 Generalized Phaser Semantics

Here we present the technical details of a task-parallel language with phasers that supports subphase blocks. We formally define the semantics of **subphase** and the phaser operations.

5.3.1 Phaser Language

The abstract task-parallel language we consider has the following synchronization syntax:

$$P ::= s \mid P_1; P_2 \mid \textbf{async}\ (\kappa)\ \{P\} \mid \textbf{finish}\ \{P\} \mid \textbf{subphase}\ \{P\}$$
$$\mid \textbf{new}\ ph \mid \textbf{signal}\ ph \mid \textbf{next} \mid \textbf{drop}\ \kappa$$
$$\kappa : \mathit{Phaser} \rightarrow \mathit{Capability}$$

where s represents a synchronization-free program and ph represents a phaser identifier.

We use the lattice *Capability* with elements $\bot < \text{SIG}, \text{WAIT} < \text{SIGWAIT}$ to encode the capability of each task to signal or await each phaser. For simplicity, we do not consider the additional capability SIGWAITNEXT used in prior work [74]. We will denote the join operator with '+,' and when $\kappa_1 \leq \kappa_2$, the expression $\kappa_2 - \kappa_1$ denotes the least element κ_3 such that $\kappa_1 + \kappa_3 = \kappa_2$. In the induced lattice $\mathit{Phaser} \to \mathit{Capability}$, the lattice operations lift point-wise to functions.

As usual, **async** spawns a new task, and a **finish** block does not complete until all tasks spawned transitively within it have completed. Phasers are allocated by **new**; for simplicity we refer to each phaser by its globally unique identifier. The task which performs the allocation is registered to the phaser in SIGWAIT mode. Capabilities of a task can be granted to child tasks via the parameter to **async**. Capabilities can be dropped by **drop**.

The phaser-specific **signal** has the effect of signaling a SIG-mode phaser at most once prior to each global **next** instruction, which then signals any remaining SIG-mode phasers and awaits all WAIT-mode phasers. The utility of a phaser-specific **signal** is in implementing split-phase barrier behavior [40].

Our addition to this typical phaser language is the **subphase** block, which modifies the behavior of **signal** and **next**. An instruction instance is said to be at *level* i if i is the depth of the enclosing subphase nest. That is, level 0 refers to code not in any subphase block, level 1 refers to code within just one subphase block, and so on. Every phaser will be associated to the level at which it was allocated.

5.3.2 Standard Semantics (No Subphases)

The standard semantics of phasers has been formally defined before [21] in a way that does not encapsulate waits inside a **next** instruction. (If the deadlock freedom

policy for signals and waits is violated, the program gets stuck.) We give a modified formalization here that utilizes the **next** instruction so that it is impossible to write a policy-violating program with respect to the ordering of signals and waits. Our formalization is designed to be highly amenable to the forthcoming subphase extension.

The globally shared state is modeled by three maps:

- K : *Task* $\rightarrow$ *Phaser* $\rightarrow$ *Capability* tracks the capability registrations and is initialized to $[(\_, \_) \mapsto \bot]$.
- S, O : *Task* $\rightarrow$ *Phaser* $\rightarrow \mathbb{N}$ track the most recent phase numbers signaled and observed by each task for each phaser. They are both initialized to $[(\_, \_) \mapsto 0]$.

For now, define the phase number increment function $adv : \mathbb{N} \rightarrow \mathbb{N}$ as $x \mapsto x + 1$. We will replace this function with a generalization in Section 5.3.6.

In the following semantics, we assume the usual asynchronous task semantics, where **async** $(\kappa)\{P_1\}; P_2$ means that the continuation, P_2, proceeds in the current task while P_1 proceeds in parallel in a newly spawned task.

We will use the notation t : *instr* to indicate that task t is performing the instruction, *instr*. There are a few synthetic instructions, not available to the programmer, that are implicitly invoked by the block structures and the compound behavior of the **next** instruction. The block-based syntactic structures are translated to synthetic instructions as follows:

- When t enters or exits a **finish** block, execute t : *enter-finish* or t : *exit-finish*, respectively.
- When t enters or exits a **subphase** block, execute t : *enter-subphase* or t : *exit-subphase*, respectively.
- When t performs **async** (κ), creating a new task, t', execute t : *spawn* κ t'.

- When t terminates, execute $t : \mathit{terminate}$.

There are four rules to manage the capability registrations. A task is initially in SIGWAIT mode for any phaser it creates; the initial phase numbers used are borrowed from other phasers the task may already be registered to. A task can drop any capability it holds. A task can grant any subset of its capabilities to a new task that it spawns; the new task inherits the parent task's S and O state. When a task terminates, it automatically drops all its capabilities.

$$\frac{x = \max O(t, \cdot)}{(K, S, O) \Rightarrow (K[(t, ph) \mapsto \text{SIGWAIT}, S[(t, ph) \mapsto x], O[(t, ph) \mapsto x])}\ t : \textbf{new}\ ph$$

$$\frac{\textbf{assert}\ \kappa \leq K(t)}{(K, S, O) \Rightarrow (K[t \mapsto K(t) - \kappa], S, O)}\ t : \textbf{drop}\ \kappa$$

$$\frac{\textbf{assert}\ \kappa \leq K(t)}{(K, S, O) \Rightarrow (K[t' \mapsto \kappa, S[t' \mapsto S(t)], O[t' \mapsto O(t)])}\ t : \mathit{spawn}\ \kappa\ t'$$

$$\frac{}{(K, S, O) \Rightarrow (K[t \mapsto [\_ \mapsto \bot]], S, O)}\ t : \mathit{terminate}$$

The next three rules give the semantics for interacting with individual phasers; the actions are signaling, observing, and awaiting. Signaling corresponds to the **signal** instruction and is also used internally by **next**. Observing and awaiting are synthetic instructions not available to the programmer and are used internally by **next**. When t signals ph, t sets its signaled phase number to one more than the observed phase number but does not modify the observed phase number; this makes multiple successive signals idempotent. When t observes ph, t increments the observed phase number. When t awaits ph, no state is updated, but the action cannot proceed until all the signaled phase numbers for ph from all SIG-registered tasks reaches at least t's current observed phase number for ph. The rule for signaling requires that the task

has correct capability.

$$\frac{\textbf{assert } \textsc{sig} \leq K(t, ph)}{(K, S, O) \Rightarrow (K, S[(t, ph) \mapsto adv(O(t, ph))], O)}\ t : \textbf{signal}\ ph$$

$$\frac{}{(K, S, O) \Rightarrow (K, S, O[(t, ph) \mapsto adv(O(t, ph))])}\ t : observe\ ph$$

$$\frac{(\min_{\textsc{sig} \leq K(x,ph)} S(x, ph)) \geq O(t, ph)}{(K, S, O) \Rightarrow (K, S, O)}\ t : await\ ph$$

The final component of the standard semantics is the global **next** operation. When task t invokes **next**, the following occur, in sequence:

1. t : **signal** ph for each ph such that $\textsc{sig} \leq K(t, ph)$.

2. t : *observe* ph for each ph such that $\bot < K(t, ph)$.

3. t : *await* ph for each ph such that $\textsc{wait} \leq K(t, ph)$.

For phasers in SIG-only mode, this definition of the **next** operation signals the phaser if it has not already been signaled since the last **next**, then observes the phaser so that a future signal will be effectual again. In WAIT-only mode, the **next** operation first observes the phaser in order to update the phase number that the task will be waiting for, then awaits this phase number to be signaled by all relevant tasks. In SIGWAIT mode, all of the above occur: the signal, the observation, and the await. To achieve deadlock freedom, the phasers are not treated one at a time; instead, all signals must occur first, then all observations and awaits.

The preceding rules for signaling, observing, and awaiting, together with the definition of **next** can be compared with the two rules for signaling and waiting in Cogumbreiro et al. [21]. In that work, observing and awaiting are performed together by the wait operation, and, when in SIG-only mode, signaling always increments, disregarding the most recent observed phase, so that observing is unnecessary.

5.3.3 Finish Semantics

The **finish** block is a convenient and structured way to await task termination. We assume the usual semantics that upon exiting a finish, a task awaits the termination of every task which was transitively spawned from within that finish block. Subject to a few additional constraints, it is known that phasers can be mixed with finishes without compromising deadlock freedom [74]. The key is to consider the dynamic nest of finish blocks and, specifically, the innermost finish block enclosing each instruction. This block is the *immediately enclosing finish* (IEF). The IEF of a phaser is the IEF of the **new** instruction that created the phaser. The rules imposed on finish blocks [74] may be formalized as follows.

Augment the state (K, S, O) as (K, S, O, F), where $F : \mathit{Phaser} \cup \mathit{Task} \to \mathbb{N}$, initially $[\_ \mapsto 0]$, remembers each phaser's IEF and the current IEF in each task. The following rules show only what happens to the F component of the state under the relevant instructions. The other state components are either preserved or updated as previously described, and F is preserved by the other instructions.

$$\frac{}{F \Rightarrow F[ph \mapsto F(t)])}\ t : \textbf{new}\ ph$$

$$\frac{\textbf{assert}\ \forall ph, (\kappa(ph) > \bot \implies F(ph) = F(t))}{F \Rightarrow F[t' \mapsto F(t)])}\ t : \mathit{spawn}\ \kappa\ t'$$

$$\frac{}{F \Rightarrow F[t \mapsto (t) + 1])}\ t : \mathit{enter\text{-}finish}$$

$$\frac{\textbf{assert}\ \forall ph, (F(ph) = F(t) \implies K(t, ph) = \bot)}{F \Rightarrow F[t \mapsto F(t) - 1])}\ t : \mathit{exit\text{-}finish}$$

Essentially there are two constraints: 1) capabilities for a phaser cannot be passed to a new task unless that phaser and the spawn point of the new task have the same IEF, and 2) a task may not retain capabilities for a phaser created within a finish block after exiting that block. It is a runtime error for either of these constraints to be violated. For convenience, the rule for *exit-finish* can alternatively be interpreted

to mean that t implicitly drops $K(t, ph)$ for the appropriate phasers for which it still has capabilities just *prior* to awaiting the termination of the enclosed tasks.

5.3.4 Subphase Level Semantics

As with **finish**, every instruction instance occurs within some dynamic nest of zero or more **subphase** blocks. The depth of this nest is relevant to the **new**, **next**, and **signal** instructions.

Definition 59. The *level* of an instruction instance is the depth of the **subphase** nest at the instruction. The *level* of a phaser is the level of the **next** instruction instance which created the phaser.

We augment the program state with one additional map, $\ell : \mathit{Phaser} \cup \mathit{Task} \to \mathbb{N}$, initially $[\_ \mapsto 0]$, giving the level of each phaser, which is set upon phaser creation, and of each task, which changes upon entry of or exit from a subphase block. Again, we will only show the effect of the relevant instructions on ℓ. The other state components are preserved or affected as previously described, and the other instructions preserve ℓ.

$$\frac{}{\ell \Rightarrow \ell[ph \mapsto \ell(t)])}\ t : \textbf{new}\ ph$$

$$\frac{}{\ell \Rightarrow \ell[t' \mapsto \ell(t)])}\ t : \mathit{spawn}\ \kappa\ t'$$

$$\frac{}{\ell \Rightarrow \ell[t \mapsto (t) + 1])}\ t : \mathit{enter\text{-}subphase}$$

$$\frac{}{\ell \Rightarrow \ell[t \mapsto \ell(t) - 1])}\ t : \mathit{exit\text{-}subphase}$$

The subphase level management is thus analogous to IEF tracking. We define the effect of ℓ on **next** and **signal** in Section 5.3.6.

Note that higher level numbers are deeper in the subphase nest. When referring to the level number we say "higher/lower" or "greater/less;" when referring to the block nesting we say "outer/inner" or "enclosing/enclosed."

5.3.5 An Inadequate Solution

A first approach to packing multiple phases of one phaser into a single phase of another phaser is to naïvely rely on the levels of the phasers and instructions. Outside of any subphase block, the **next** instruction would affect phasers which were created outside of any subphase block; at level 1, **next** would affect level 1 phasers, and so on. In general, t : **next** would interact with every phaser ph such that $\ell(ph) = \ell(t)$.

However, this approach is not sufficient because it does not account for the phaser's need to know the SIG-registered tasks it should expect a signal from. Suppose only a subset of the registered tasks enters a **subphase** block to interact with a level 1 phaser, while other registered tasks remain at level 0. The fact that some tasks elect to remain at level 0 needs to count as if those tasks were signaling the level 1 phaser, repeatedly, until all registered tasks are out of the **subphase** block. Therefore, **next** instructions executed by both the level 0 and level 1 tasks must signal the level 1 phaser in some way.

We conclude that a t : **next** instruction must interact with phasers created at level $\ell(t)$ *or greater*.

5.3.6 Non-integer Phase Number Semantics

Ordinarily, a phaser maintains an internal phase count, and each task remembers the last phase number it signaled or observed for each phaser. We replace these single counters with lists of counters, one for each subphase level. Advancing a phase number at level i involves incrementing the ith counter and resetting the jth counter to zero for all $j > i$. The counter lists are compared lexicographically for the purpose of awaiting a phase. In this way, advancements of the phase number (signals) occurring at levels greater than i do not release waits occurring at levels less than or equal to i. Conversely, a single signal at level i has the effect of arbitrarily many signals at levels greater than i. Recall that it is by this mechanism that all the **next** instructions in

line 10 and line 12 of Listing 5.6 are not sufficient to release the **next** in line 20.

5.3.6.1 Strings of Unbounded Digits

To incorporate the idea of this more complex phase number representation into the existing semantics, we define basic arithmetic operations using lists of counters. One may think of the list of counters as a string of digits in an esoteric number system where the radix is infinite. Instead of drawing digits from the integer range $[0, b)$ for some finite radix b, we let each digit be an unbounded natural number.

Definition 60. Let $\mathbb{N}^*$ be the set of strings of natural numbers. We notate such strings as $\underline{x_0}\,\underline{x_1}\cdots\underline{x_n}$, underlining individual digit values. Let x_i denote the ith digit in the string x, or 0 if $|x| \leq i$. The zero value is representable as the empty string, ε, or strings of zeros such as $\underline{0}$ or $\underline{0}\,\underline{0}$. Addition, lexicographical comparisons, and truncation are given as follows:

$$
\begin{aligned}
x + y &\triangleq \underline{(x_0 + y_0)}\,\underline{(x_1 + y_1)}\cdots \\
x < y &\triangleq \exists j \;.\; (\forall i < j \;.\; x_i = y_i) \wedge x_j < y_j \\
x = y &\triangleq \forall i \;.\; x_i = y_i \\
x_{0:j} &\triangleq \underline{x_0}\,\underline{x_1}\cdots\underline{x_j}
\end{aligned}
$$

One may understand this number system by interpret ting the digits as the coefficients of a polynomial evaluated at the formal infinitesimal, ϵ. For example, $\underline{0}\,\underline{2}\,\underline{0}\,\underline{1}$ becomes $2\epsilon + \epsilon^3$, which is strictly less than $\underline{1}\,\underline{1}$, which becomes $1 + \epsilon$.

We are now able to say exactly what is meant when we say that **next** uses phase number increments whose size depends on the subphase level. Instead of manipulating phase numbers by adding one, we manipulate them by adding $\underline{0}^i\underline{1}$, where i is the present instruction level. Phase numbers are subsequently rounded to $i + 1$ digits of precision using truncation: $x_{0:i}$. The combination of increment and truncation is

perceived as rounding up.

5.3.6.2 Unsuitability of Real-valued Phase Numbers

The crucial property of our arithmetic over $\mathbb{N}^*$ that we depend on is that every multiple of $\underline{0}^i\underline{1}$ is strictly less than $\underline{0}^{i-1}\underline{1}$. This corresponds precisely to the fact that any number of signals occurring inside a **subphase** block is insufficient to release a waiter outside the block, and, conversely, a single signal outside the block repeatedly affects all waiters within the block indefinitely.

The preceding encoding of non-integer phaser numbers is more suitable than using rational or real numbers, which lack the required property. Since it is not known up-front how many **next** instructions will be executed within a single **subphase** block, we cannot select an appropriate fractional increment.

5.3.6.3 Modified Phaser Semantics

We are now ready to adjust the semantics of phasers to accommodate subphase levels. Having set up the preliminary semantics carefully, the change is minimal.

Change the types of the maps S and O to $\mathit{Task} \to \mathit{Phaser} \to \mathbb{N}^*$, and initialize them to $[(\_, \_) \mapsto \varepsilon]$. Redefine the increment function, *adv* (used in t : **signal** ph and t : *observe* ph), from $x \mapsto x + 1$ to $x \mapsto x_{0:i} + \underline{0}^i\underline{1}$, where $i = \ell(t)$.

Finally, re-define **next** to be sensitive to the levels by ignoring phasers from outer levels. That is, upon t : **next**, all three actions (signal, observe, await) are *skipped* for each ph such that $\ell(ph) < \ell(t)$.

5.4 Properties

Theorem 61. *Absent any* **subphase** *blocks, the semantics reduces to the original semantics for ordinary phasers.*

Proof. The value of $\ell(\cdot)$ is zero for every task and every phaser. The increment used by *adv* in **signal** *ph* and *observe ph* is, therefore, always $\underline{1}$, and truncation has no effect. That is, for $i = 0$, $x \mapsto x_{0:i} + \underline{0}^i\underline{1}$ reduces to the integer increment function, $x \mapsto x + 1$. □

5.4.1 Deadlock Freedom (Without Finish)

The semantics is carefully defined so that, for a given task t, the values of $O(t, \cdot)$ are almost in agreement across all phasers. This facilitates the proof of deadlock freedom; however, the details are tedious because the agreement is not perfect, due to subphases. Whenever the values differ, we can find a small bound for the discrepancy that is dependent on the subphase level of the phaser with the smaller value:

Lemma 62. *Let t be a task registered in any modes to phasers ph_1 and ph_2. Suppose t is not in the middle of the observation portion of a* **next**. *Without loss of generality, assume $O(t, ph_1) \leq O(t, ph_2)$. Then there is an $x < \underline{0}^{\ell_1}\underline{1}$, where $\ell_1 = \ell(ph_1)$ is the subphase level of ph_1, such that $O(t, ph_1) + x \geq O(t, ph_2)$.*

Proof. By induction, examining the most recent instruction that set the value of $O(t, ph_1)$ or $O(t, ph_2)$. We have the following cases:

1. ph_2 already exists, and t just executed **new** ph_1: Then $O(t, ph_1) = \max O(t, \cdot)$. Hence $O(t, ph_1) \geq O(t, ph_2)$. But we assumed without loss of generality that $O(t, ph_1) \leq O(t, ph_2)$. Hence $O(t, ph_1) = O(t, ph_2)$. Thus, choose $x = \varepsilon$ (the zero value).

2. ph_1 already exists, and t just executed **new** ph_2: Then $O(t, ph_2) = \max O(t, \cdot)$. Hence, there already exists a phaser ph_3 (possibly ph_1) such that $O(t, ph_2) = O(t, ph_3)$. If $ph_3 = ph_1$, again choose $x = \varepsilon$. Otherwise, choose x by appealing to the I.H. on the phasers ph_1 and ph_3.

3. ph_1 and ph_2 already exist.

(a) t was just spawned by an **async**, inheriting capabilities for ph_1 and ph_2 from the parent task t'. Then $O(t, \cdot) = O(t', \cdot)$, so appeal to the I.H. on t'.

(b) t just executed the observation portion of a **next** instruction. Put $i = \ell(t)$, $\ell_1 = \ell(ph_1)$, and $\ell_2 = \ell(ph_2)$. Let x' be the value of x obtained from the I.H. prior to executing this **next**. So we know that $x' < \underline{0}^{\ell_1}\underline{1}$.

i. If $i \leq \ell_1, \ell_2$, then $O(t, ph_1)$ and $O(t, ph_2)$ were both truncated to position i and were both incremented by $\underline{0}^i\underline{1}$. This erases the x' discrepancy (since $i \leq \ell_1$) and ensures $O(t, ph_1) = O(t, ph_2)$, so choose $x = \varepsilon$.

ii. If $i > \ell_1, \ell_2$, then $O(t, ph_1)$ and $O(t, ph_2)$ remain unchanged, so choose $x = x'$.

iii. If $\min\{\ell_1, \ell_2\} < i \leq \max\{\ell_1, \ell_2\}$, then $O(t, \cdot)$ increases for one of the phasers and remains constant for the other. We assume, without loss of generality, that it increases for ph_2. We deduce that $\ell_1 < \ell_2$. The value of $O(t, ph_2)$ is truncated to position i and then increased by $\underline{0}^i\underline{1}$. Therefore choose $x = \underline{0}^i\underline{1}$ so that $O(t, ph_1) + x$ meets or exceeds the updated value of $O(t, ph_2)$. Finally, $i > \ell_1$, so $x < \underline{0}^{\ell_1}\underline{1}$, as desired.

□

Lemma 63. *Suppose task t is in the awaiting portion of a* **next** *and is blocked on phaser ph. Then there exists a task, t', registered in at least* SIG *mode to ph such that $O(t, ph) > S(t', ph)$. Moreover, there exists an index $j \leq \ell(ph)$ such that $O(t, ph)_j > S(t', ph)_j$.*

Proof. That a task t' exists with $O(t, ph) > S(t', ph)$ is evident from the semantics of t awaiting ph, which computes a minimum over $S(\cdot, ph)$ on the set of signalers of ph. Since a **next** at some level i does not interact with phasers of subphase levels strictly less than i, then $O(t, ph)_i = S(t', ph)_i = 0$ for all $i > \ell(ph)$. It remains that

the nonzero difference between $O(t, ph)$ and $S(t', ph)$ occurs in at least one digit at a position less than or equal to $\ell(ph)$. □

The following proof of deadlock freedom works by deriving a contradiction from the existence of a cycle. The contradiction essentially identifies a decreasing cycle of $\mathbb{N}^*$ numbers, $O(t_i, ph_i) > S(t_{i+1}, ph_i) = O(t_{i+1}, ph_i) \stackrel{?}{=} O(t_{i+1}, ph_{i+1})$ for all i modulo n. However, the proof is complicated by the fact that the numbers are not actually always decreasing; they sometimes increase (indicated by the $\stackrel{?}{=}$). We are saved by Lemma 62 and Lemma 63, which show that the decreases are larger than the increases.

Theorem 64. *Subphaser programs without* **finish** *blocks are deadlock-free.*

Proof. Suppose there exists a cycle of length n, wherein for each i modulo n,

1. task t_i is awaiting phaser ph_i;
2. task t_{i+1} is the task t' given by Lemma 63.

Since t_{i+1} is a signaler of ph_i, and since t_{i+1} is awaiting a phaser, t_{i+1} must be currently executing a **next** and have already signaled and observed ph_i. Hence, $S(t_{t+1}, ph_i) = O(t_{i+1}, ph_i)$.

Examine the cycle of values for i modulo n:

$$O(t_i, ph_i) > S(t_{i+1}, ph_i) \tag{5.1}$$

$$= O(t_{i+1}, ph_i) \tag{5.2}$$

$$\stackrel{?}{=} O(t_{i+1}, ph_{i+1}) \tag{5.3}$$

The degree of decrease and increase at each step is tied to the subphase level, $\ell(ph_i)$. The decrease in Eq. (5.1) involves a decrease in digit $\ell(ph_i)$ (Lemma 63) and the change in Eq. (5.3), if it is an increase, does not affect digits past position $\ell(ph_i)$ (Lemma 62).

Therefore, let $\ell^* = \min_i \ell(ph_i)$ be the least subphase level among all the phasers in the cycle. Now the sequence $O(t_i, ph_i)_{\ell^*}$ forms a cycle of digits that is strictly decreasing, which is a contradiction. □

Remark 65. *When t creates a new phaser ph, the initial choice of $O(t, ph)$ is $\max O(t, \cdot)$. This decision is solely to make the proofs convenient. The values of $O(t, \cdot)$ for different phasers interact with one another in the proofs, but not in the semantics. Therefore, the implementation is free to make a simpler choice without losing the deadlock-freedom guarantee: initialize $O(t, ph)$ to zero.*

5.4.2 Relationship to Finish

The preceding deadlock-freedom guarantee is formulated to consider only phasers. We can introduce **finish** blocks without causing deadlocks using the same approach as for standard phasers [74] that relies on the immediately enclosing finish rules.

Theorem 66. *The introduction of* **finish** *block delimiters cannot introduce a deadlock under the IEF rules in Section 5.3.3.*

Proof. Suppose a program is initially deadlock-free and that a **finish** block is introduced. When task t reaches the end of the **finish**, it awaits the termination of all the tasks which spawned transitively within the **finish**. By the rule for t : *exit-finish*, t has dropped capabilities for all phasers which it created inside the **finish**. By the rule for t : *spawn* κ t', where t' is any task created *within* the **finish**, the set of awaited tasks cannot hold capabilities for any phaser created *outside* the **finish**. Therefore, the set of awaited tasks can only have blocking synchronization dependences among one another, not involving any tasks external to the **finish** and no longer involving t. Hence, the await for task termination cannot be part of a cycle, following from the tree-structured nesting of **finish** blocks, as proved by Lee and Palsberg [57]. □

5.5 Algorithms

We give algorithms for implementing the semantics described in the preceding section.

5.5.1 Construct Management

Task, phaser, and capability management algorithms are presented in Algorithm 11. State is maintained in six globally shared maps:

- $\mathsf{cap} : \mathit{Task} \to \mathit{Phaser} \to \mathit{Capability}$
- $\mathsf{level} : \mathit{Phaser} \cup \mathit{Task} \to \mathbb{N}$
- $\mathsf{signaled}, \mathsf{observed} : \mathit{Task} \to \mathit{Phaser} \to \mathbb{N}^*$
- $\mathsf{phase} : \mathit{Phaser} \to \mathbb{N}^*$
- $\mathsf{parties} : \mathit{Phaser} \to \mathcal{P}(\mathit{Task})$

The first four of these correspond directly to maps we saw before in the semantics. The phase map will store the current minimum value in signaled for each phaser. The parties map will be used to store the set $\{t \mid \text{SIG} \leq \mathsf{cap}(t, ph)\}$ for each phaser ph. There is also a task-local variable, task_{cur}, to store the identifier of the current task.

The procedure $\textsc{Async}(\kappa, f)$ initializes a new task to execute f with initial capabilities κ. We require that these capabilities already be held by the current task (line 4). The new task is set up with the same signaled and observed phase numbers that the current task sees for each relevant phaser (lines 8–11), as well as the same subphase level as the current task (line 7). Note that there is no need to initialize the signaled value for a phaser for which the task does not have the SIG capability, since this value is only used by the signal action. The new task is added to the parties set of each phaser for which it will have the SIG capability (line 12). Once the new task

Algorithm 11 Phaser Implementation, Part 1

```
1: ▷ cap, parties, level, signaled, observed are global variables
2: ▷ task_cur is a task-local variable
3: procedure ASYNC(κ : Phaser → Capability, f)
4:     assert κ ≤ cap(task_cur)
5:     t ← new task id
6:     cap(t) ← κ
7:     level(t) ← level(task_cur)
8:     for ph ∈ {ph | ⊥ < κ(ph)} do
9:         observed(t, ph) ← observed(task_cur, ph)
10:    for ph ∈ {ph | SIG ≤ κ(ph)} do
11:        signaled(t, ph) ← signaled(task_cur, ph)
12:        atomically parties(ph) ← parties(ph) ∪ {t}
13:    do asynchronously
14:        task_cur ← t
15:        f()
16:        DROP(cap(t))
17:    return t

18: procedure SUBPHASE(f)
19:    level(task_cur) ← level(task_cur) + 1
20:    f()
21:    level(task_cur) ← level(task_cur) − 1

22: procedure NEW()
23:    ph ← new phaser id
24:    cap(task_cur, ph) ← SIGWAIT
25:    level(ph) ← level(task_cur)
26:    signaled(task_cur, ph) ← ε
27:    observed(task_cur, ph) ← ε
28:    phase(ph) ← ε
29:    parties(ph) ← {task_cur}
30:    return ph

31: procedure DROP(κ : Phaser → Capability)
32:    assert κ ≤ cap(task_cur)
33:    cap(task_cur) ← cap(task_cur) − κ
34:    for ph ∈ {ph | SIG ≤ κ(ph)} do
35:        atomically parties(ph) ← parties(ph) \ {task_cur}
36:        UPDATEPHASE(ph)
```

Algorithm 12 Phaser Implementation, Part 2

```
1:  procedure ADV(y, x)
2:      l ← level(task_cur)
3:      for i ← 0 ... l − 1 do
4:          y_i ← x_i
5:      y_l ← x_l + 1
6:      return y

7:  procedure SIGNAL(ph)
8:      assert SIG ≤ cap(task_cur, ph)
9:      ▷ If the following copy is not atomic, intermediate states must not exceed the
        final state. For example, the copy may proceed right-to-left.
10:     signaled(task_cur, ph) ← ADV(observed(task_cur, ph))
11:     UPDATEPHASE(ph)

12: procedure OBSERVE(ph)
13:     observed(task_cur, ph) ← ADV(observed(task_cur, ph))

14: procedure UPDATEPHASE(ph)
15:     ▷ Concurrent updates to parties(ph) and to signaled(·, ph) can arise; they must
        be sequentially consistent
16:     x ← min_{t∈parties(ph)} signaled(t, ph)
17:     atomically phase(ph) ← max(x, phase(ph))

18: procedure WAIT(ph)
19:     while phase(ph) < observed(task_cur, ph) do
20:         sleep until phase(ph) changes

21: procedure NEXT()
22:     L ← {ph | level(ph) ≥ level(task_cur)}
23:     for ph ∈ {ph | SIG ≤ cap(task_cur, ph)} ∩ L do
24:         SIGNAL(ph)
25:     for ph ∈ {ph | ⊥ < cap(task_cur, ph)} ∩ L do
26:         OBSERVE(ph)
27:     for ph ∈ {ph | WAIT ≤ cap(task_cur, ph)} ∩ L do
28:         WAIT(ph)
```

has finished executing f, it drops all its capabilities, which, by this point, may be different from κ (line 16).

SUBPHASE(f) executes f within a subphase block by incrementing the current level before f and decrementing the current level afterward.

Creation of a phaser in NEW() consists of initializing the six fields. The phaser's level is the current subphase level. The current task is the only member of the parties set for the new phaser since it alone has the SIG capability for the phaser.

The DROP(κ) procedure removes the given capability registrations from the current task (line 33), requiring that the task did indeed hold them (line 32). For all phasers for which the task is dropping the SIG capability, the task is removed from the phaser's parties set (line 35). This operation must be done atomically since two tasks may concurrently drop the capability. Finally, the phase value for such phasers is recomputed since there are fewer SIG tasks remaining (line 36). The UPDATEPHASE algorithm is given in the next section.

5.5.2 Phaser Operations

The phaser synchronization operations and the internal procedures that affect phase numbers are given in Algorithm 12.

The ADV(x) procedure creates a new list of counters that differs from x by adding one to the lth counter, where l is the current subphase level, and by leaving off the counters past l (which means they are implicit zeros). In an implementation, the storage for the return value may already be allocated; if there are more than $l + 1$ counters available in the list, the extra ones should be reset to zero or destroyed. In this case, it is very important that an intermediate view of the output not exceed the final value; that is, the resetting of extra counters must happen before the increment of the lth counter. A violation of this requirement would allow the invocation of ADV in SIGNAL line 10 to break a concurrent min computation in UPDATEPHASE line 16.

The SIGNAL(ph) procedure requires the SIG capability (line 8) and updates the signaled value for the ph based on the current observed value, as in the semantics. Since this operation may release the phaser, we recompute the phase number (line 11).

OBSERVE(ph) is an internal procedure used by NEXT that advances the observed value in place.

UPDATEPHASE(ph) is the internal procedure that is called whenever the current phase number of ph may have changed. This happens upon any signal or drop of the SIG capability for ph. The computation takes a min over all SIG-capable tasks, as indicated by the parties field (line 16). It is possible that the set of parties and the values read from signaled are being concurrently updated. This is tolerable because these concurrent tasks will also invoke UPDATEPHASE. However, when considering a weak memory consistency model, it is very important that at least one of those tasks reads the most up-to-date view of this data; otherwise all tasks could fail to advance the phase number. When the new phase value is stored, we atomically ensure that we are not decreasing the existing value (line 17), which might otherwise have occurred due to the concurrent invocations of this procedure. In our Java implementation, we achieve the preceding requirements for consistency and atomicity by acquiring a phaser-specific lock for the duration of the procedure. This same lock is acquired by modifiers of parties(ph). (Relevant modifiers of signaled acquire this lock naturally by subsequently calling UPDATEPHASE.)

The WAIT(ph) procedure repeatedly checks the phase field, updated by others, until it reaches at least the current observed value.

The NEXT() procedure performs the expected signal-observe-wait operations for each relevant phaser. All the signals must occur first (lines 23–24). This procedure differs from a standard implementation of **next** in that phasers are ignored if their level is less than the current subphase level (line 22).

5.6 Evaluation

To demonstrate the effectiveness of the subphase extension to phasers, we compared the performance of four phaser benchmark programs. Each benchmark is implemented in at least three variants:

- B (baseline, task-aware phasers): uses direct **signal** and **wait** operations on individual phasers
- H (Habanero phasers): uses the global **next** operation
- S (Habanero phasers with subphases): uses **next** together with **subphase** blocks

The use of global **next** operations automatically guarantees deadlock freedom for the H and S variants; the deadlock-freedom of the B variant must be determined by inspection.

We consider the use of per-phaser signals and waits to be the baseline of comparison because it affords precise control by the programmer over the phaser synchronization. In general, the coarse granularity of phaser actions imposed by the use of **next** means that the H variant is over-synchronized compared to the B variant. Our novel subphase-based approach is effective as an improved deadlock-freedom policy for phaser usage if the S variant out-performs just the H variant, thereby recovering efficiency that is lost by using the traditional deadlock-freedom approach. One might not expect the S variant to be able to out-perform the baseline B variant; however, this can indeed happen and illustrates a fundamental advantage of organizing phasers using subphases.

5.6.1 Benchmarks

Iterative Averaging. The simplest benchmark we consider is the iterative averaging program discussed in Section 5.2.2 and in prior work [19, 75]. Each cell in a

1D array is iteratively updated to the average of its two neighbors. Worker tasks synchronize on a common phaser twice per iteration: once to announce that each has read its neighbors' old data and once to announce that each has updated its own data. A second phaser is used to alert the root task that the workers have terminated.

- B: Each worker interacts with the termination phaser only once, by dropping it upon termination. Therefore, the root task needs only to await the termination phaser once.

- H: The workers are not able to interact with the two phasers independently, and, therefore, they act on the termination phaser $2i + 1$ times, where i is the number of iterations. Consequently, the root task must pump the termination phaser $2i + 1$ times as well.

- S: By locating the workers' common phaser in a subphase level deeper than the termination phaser, the workers are able to synchronize with each other without acting on the termination phaser. Therefore, the synchronization pattern is identical to the B variant.

Point-to-Point Iterative Averaging. This benchmark performs the same computation as the Iterative Averaging benchmark, but with a key synchronization difference. The workers no longer use a single shared phaser to coordinate the data reads and writes. Instead, there is a unique phaser associated to each worker, which is registered to the phaser in SIG-only mode and to its two neighboring phasers in WAIT-only mode. This point-to-point pattern removes the unneeded bottleneck of a phaser shared by all the tasks.

- B: As before, the termination phaser is needs only be awaited by the root task once.

- H: As before, the root task must pump the termination phaser $2i + 1$ times, where i is the number of worker iterations.
- S: All the workers' phasers are located at a deeper subphase level than the termination phaser so that, as before, the root task needs only await the termination phaser once.

Inverse Iteration Conjugate Gradient. Inverse iteration (II) finds an eigenvector for a symmetric positive-definite matrix by solving a series of linear systems. The solution vector of each system is normalized in parallel by a set of workers tasks and then becomes the constant term in the next system to solve. Each linear system is solved using the iterative conjugate gradient (CG) method, which is itself parallelized among a set of workers. A Fortran benchmark with a similar II CG algorithm can be found the NAS Parallel Benchmark Suite [6]. In a departure from that benchmark, in our benchmark the CG workers and the associated phasers are created and destroyed upon each invocation of the CG subroutine in order to attempt a modular separation between the II and CG parts of the programs. However, even this will not be enough to prevent unwanted interactions between their phasers. The fact that the II implementation and the CG subroutine are both independently parallelized is a natural use case for subphases to prevent unnecessary synchronization.

- B: With precise control over the phaser actions, we can have uni-directional synchronization actions from each II worker to the root task and vice versa (using a pair of phasers in different modes), uni-directional synchronizations between each CG worker and the root task (another pair of phasers), and one instance of all-to-all synchronization among the CG worker tasks (one phaser in SIGWAIT mode).
- H_1: With a pair of phasers in different modes, a global **next** forces every interaction between workers and the root to be a bi-directional synchronization since

both the SIG phaser and the WAIT phaser are operated on. Therefore, we use only a single phaser in SIGWAIT mode to coordinate with the II workers, and another single phaser in SIGWAIT mode to coordinate the CG workers. A highly undesirable consequence of this is that the root task advances the II phaser in lockstep with the CG phaser. Therefore, each of the II workers must pump the II phaser for as many iterations as the CG subroutine takes, which is not a fixed number.

- H_2: A second H variant is possible. One cost of using SIGWAIT phasers in H_1 instead of separate SIG and WAIT phasers is that the CG workers always synchronize all-to-all, even though it usually suffices to synchronize with the root task in all-to-one and one-to-all patterns. We therefore return to using a pair of separate SIG and WAIT phasers for CG, as in the B variant. The cost of this alternative is that when all-to-all synchronization is actually required, all phasers must be advanced twice.

- S_1: We improve on H_1 in two ways. First, the entire CG subroutine is enclosed in a subphase block so that II workers do not need to pump the II phaser during CG computations. Second, when the CG workers need only to synchronize all-to-all with one another, excluding the root, they enter an additional subphase block, which the root omits.

- S_2: We improve on H_2 in two ways. Again we enclose CG in a subphase block. Second, all-to-one synchronization from CG workers (signaling) to the root (waiting) is enclosed in a subphase block, while one-to-all synchronization from the root to the workers is not. This prevents the workers from synchronizing with each other when it is not necessary.

There are more variants of this benchmark that could be explored, especially by adjusting the II synchronization further. However, we have focused mostly on the

CG subroutine because it is the most intensive part of the computation.

QR Iteration. The final benchmark is the program presented in Section 5.2.3, which solves a linear system through repeated QR factorizations [29, 70]. The architecture of this program differs from the preceding CG benchmark in that the tasks which perform QR factorization in parallel live for the duration of the program, whereas the tasks which performed CG in parallel are created and destroyed upon each invocation of the CG subroutine. By utilizing a persistent set of worker tasks, we avoid the cost of repeatedly setting up new tasks and registering them to new phasers. Each QR worker can also store its matrix partition information, which is constant throughout the program, in local memory, rather than requiring this information to be factored out into shared memory. The persistent-worker architecture is key to importance of the QR benchmark. The synchronization in the H variants of preceding programs could be made to resemble that of the B variants through the use of finish blocks, which are designed to await task termination. But finish blocks are unsuitable for the present benchmark and cannot be used to prune unwanted synchronization.

- B: One phaser advances twice upon every QR iteration, once when the workers have completed the iteration and again when the root has completed the iteration. A second phaser, used only by the workers, ensures that they eliminate the matrix entries at the appropriate stages. After a worker has eliminated all of its entries, it continues to pump the stage phaser until all workers have eliminated their entries.

- H: Since the workers cannot separately advance the two phasers, the root task must pump the stage phaser as well.

- S_1: By locating the stage phaser interactions in a subphase block (see Listing 5.7), the root task no longer needs to pump it. Interestingly, a worker that

eliminates all of its entries early *also* no longer needs to continue pumping the stage phaser. By exiting the subphase block and issuing **next**, it effectively drops the stage phaser *temporarily*—a technique that is in no way possible to achieve using finish blocks. This improvement means that the S_1 variant exhibits less unnecessary synchronization than even the B variant.

- S_2: The two phasers are conflated into a single phaser, which, as discussed in Section 5.2.3, has the same effective synchronization semantics. However, due to the implementation, during a single **next**, the root task is repeatedly awakened upon each **next** of each worker, even those in a subphase block, to test whether the phase number has reached the desired value.

5.6.2 Experimental Setup

All benchmarks are implemented in Java and were run under the OpenJDK 10 VM in a Linux environment on an 8-core Intel i7 processor. New tasks are always immediately scheduled for execution on a thread pool that grows dynamically to accommodate the number of concurrent tasks. Two implementations of phasers are used, a traditional phaser for the B and H variants, and an extended phaser for the S variant, which supports subphases. The two implementations share a common algorithmic core, differing only in whether the phase number is represented as a single integer or an array of integers. (Recall that in the absence of any subphase blocks, the array of integers would have length one, and the two implementations would effectively be equivalent. However, we chose not to evaluate the B and H variants using the extended phaser as its array-handling code might have introduced unfair overhead.)

For each variant of each benchmark, we collected information about the deterministic properties of the code. These properties are the total numbers of tasks and phasers created, the number of task-phaser pairs for which the task is registered to the phaser with any capability, and the number of times a phaser's signal and wait

methods are invoked. In the case of **next**, the underlying signal and wait interactions with each phaser are counted individually.

We then took measurements of two performance metrics: the total number of times a task must actually block in order to await a phase and the execution time. These measurements are averaged over 30 runs on the same virtual machine instance, following 5 warm-up runs, which is a standard procedure for mitigating JIT compilation noise [34]. The number of blocks is nondeterministic and may be less than the number of waits since a task may arrive at a phaser late, after every other signaling party. For S variants, the number of blocks may also *exceed* the number of waits. In the implementation, while a task awaits an outer phase, each concurrent signal of an inner phase causes that task to awaken and re-check the phase number. (A more advanced implementation, which we did not pursue, could selectively awaken only those tasks waiting on the same subphase level as the signal.)

5.6.3 Discussion

The deterministic properties of the benchmark variants are given in Table 5.1. The performance measurements (number of blocks and execution time) are given in Table 5.2. Fig. 5.8 renders the execution times, which are reported with a 95% confidence interval. We highlight some notable features of the results for each benchmark.

As expected for Iterative Averaging from the discussion in Section 5.2.2, the constraints imposed by the traditional deadlock-freedom policy for phasers causes a large increase in unnecessary synchronization over the baseline, including double the number of signals. However, the use of subphases allows us to recover exactly the baseline synchronization semantics. In practice, the H variant (24.29 seconds) is slower than both the B (22.71 seconds) and S (23.39 seconds) variants. The S variant recovers some, but not all, of the time lost by the H variant. Despite exhibiting the same synchronization actions as the baseline, the subphase variant may owe its slightly

Table 5.1: Phaser benchmark properties.

Benchmark	Variant	Tasks	Phasers	Reg.	Signals	Waits
Iterative Averaging	B	9	2	18	3 200 000	3 200 001
	H	9	2	18	6 400 000	3 600 001
	S	9	2	18	3 200 000	3 200 001
P2P Iterative Avg.	B	9	11	43	3 200 000	6 400 001
	H	9	11	43	6 400 000	6 800 001
	S	9	11	43	3 200 000	6 400 001
Inverse Iteration CG	B	12145	2276	38692	571 476	582 088
	H_1	12145	759	12903	1 947 390	1 947 457
	H_2	12145	1517	25789	2 349 684	2 349 684
	S_1	12145	759	12903	960 866	960 912
	S_2	12145	1517	25789	1 152 048	849 228
QR Iteration	B	8	2	16	234 384	234 384
	H	8	1	8	246 720	246 720
	S_1	8	2	16	191 208	191 208
	S_2	8	1	8	162 424	162 424

Table 5.2: Phaser benchmark performance measurements. Execution times are given with a a 95% confidence interval.

Benchmark	Variant	Blocks	Time (s)
Iterative Averaging	B	2 799 995	22.71 ± 0.07
	H	3 199 922	24.29 ± 0.07
	S	2 799 997	23.39 ± 0.04
P2P Iterative Avg.	B	1 710 831	9.26 ± 0.09
	H	2 137 780	11.96 ± 0.08
	S	1 707 237	9.40 ± 0.10
Inverse Iteration CG	B	568 716	4.53 ± 0.05
	H_1	1 757 229	9.74 ± 0.05
	H_2	1 691 778	10.68 ± 0.03
	S_1	932 174	7.66 ± 0.01
	S_2	615 646	5.26 ± 0.09
QR Iteration	B	201 487	1.52 ± 0.05
	H	215 879	1.76 ± 0.00
	S_1	137 750	0.98 ± 0.02
	S_2	183 029	1.55 ± 0.04

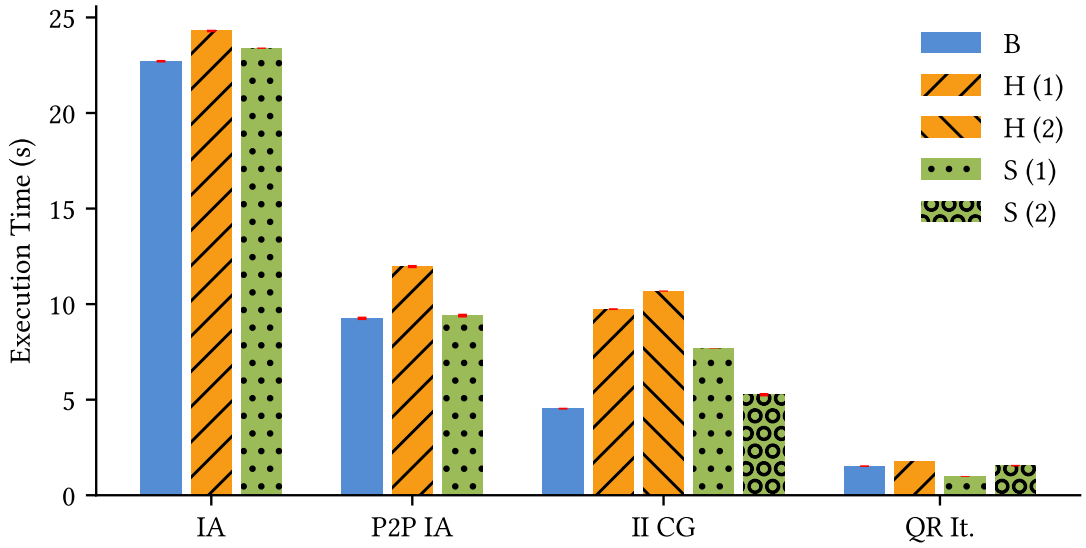

Figure 5.8: Execution times for phaser benchmarks in three styles showing the mean with a 95% confidence interval (red).

reduced performance to the more complex phaser implementation it requires.

In the Point-to-Point Iterative Averaging benchmark we see a similar effect as in the (all-to-all) Iterative Averaging benchmark. Though there is a two-fold increase in the number of waits per iteration, as each task awaits its two neighbors individually, the total execution time is dramatically reduced in all three variants to between 9 and 12 seconds. This is an expected result since the synchronization dependences are more precise here. The same execution-time relationship holds among the three variants as before: using a global **next** operation causes over-synchronization that slows down the execution, but this effect is almost entirely mitigated by the use of subphases (9.40 seconds), very closely approaching the baseline (9.26 seconds).

The inefficiency of the deadlock-freedom policy for traditional phasers becomes even more pronounced for Inverse Iteration Conjugate Gradient. Whereas in the Iterative Averaging case only the root task needed to pump a phaser, in the present case all eight Inverse Iteration workers must pump a phaser. In both H variants we

find the execution time is more than double the baseline. By utilizing subphases, in S_1 and S_2 we can reduce the number of signals and waits by half compared to the respective H_1 and H_2 variants; however we cannot fully recover the economy of the baseline. The payoff of reducing the number of signals and waits by half results in an execution time also reduced by half in the S_2 variant compared to H_2. The best subphase variant (S_2) achieves 5.26 seconds compared to the baseline's 4.53 seconds.

Finally, QR Iteration demonstrates a scenario where the number of actual blocks can exceed the number of waits due to the implementation of waiting on subphases. Recall that S_2 uses a single phaser for all of the synchronization, relying on different subphase levels to separate workers synchronizing with workers from workers synchronizing with the root task. In contrast, S_1 factors the synchronization into two separate phasers at distinct subphase levels. This difference means that in S_2, the root task is repeatedly awakened every time the workers synchronize with each other, while in S_1 the root task is undisturbed until a worker exits the subphase and signals the second phaser. The number of blocks in S_2 exceeds the number of waits by over 20,000, while in S_1 the number of blocks is less than the number of waits by over 53,000. It is not surprising, then, that S_1 (0.98 seconds) outperforms S_2 (1.55 seconds). What may be surprising is that S_1 even outperforms the baseline (1.52 seconds). The reason for this is one of the most compelling arguments in favor of subphases. We have noted that some of the synchronization problems in the preceding benchmarks can be efficiently solved using finish blocks, which help to modularize the code and insulate unrelated phasers from each other. Finish blocks, moreover, are compatible with the traditional phaser deadlock-freedom policy. However, one crucial action that cannot be achieved by finish blocks is the *effectively temporary* dropping of a phaser. That is what we are after in the QR Iteration program. We would like workers which complete a round early to temporarily drop out of the iterative synchronization with other workers until the next QR round begins. Even with precise control over the individual signals and

waits that is possible in the baseline, we still cannot have a worker drop a phaser only to re-register again later. (Doing so in a safe, non-racy way is another avenue of research.) However, with subphases we can achieve the desired effect anyway: A worker which completes a round early can exit the subphase and round up the phase number. Its interactions with other workers that remain in the subphase block are thus suppressed for a time. Altogether, both S variants of QR Iteration are, therefore, able to exhibit fewer signals and waits than the baseline, and S_1 exhibits over 63,000 fewer blocks than the baseline, resulting in a 36% reduction in execution time.

5.7 Related Work

Phasers in the form that we discuss were introduced by Shirako et al. [74] into the Habanero-Java language [15] as a successor to X10 clocks [17] and barriers [66], especially the fuzzy barrier [40]. These phasers are not susceptible to deadlock, owing to the semantics of the global **next** operation.

Phasers can be seen as a generalization of the familiar cyclic barrier construct that is found, for example, in Java [68] and OpenMP [66]. A barrier waits for some fixed number of tasks to arrive and wait, at which point all these tasks are released. In this sense, tasks always interact with barriers as if in SIGWAIT mode. If the barrier supports repeated invocations of this arrive-and-release procedure, the barrier is called cyclic. Barriers do not have a global operation, like **next**, for simultaneously arriving at all barriers to which a task is registered. Therefore, barriers can be deadlocked.

X10 clocks are a synchronization mechanism that is simpler than phasers in the sense that every registered task must have both SIG and WAIT capabilities (in this way a clock resembles a cyclic barrier); however, clocks have a global **next** operation and are, therefore, not susceptible to deadlocks (unlike cyclic barriers) [17].

The java.util.concurrent.Phaser class found in Java [69] does not represent the full definition of a phaser found in the literature. In particular, specific tasks are not

registered to phasers; only a total count of signaling parties is maintained. There is no capability management, and there is no global **next** operation. Therefore, the Java Phaser is susceptible to deadlock in the same way that cyclic barriers are. As a result, there is a need for a tool like Armus [19], a dynamic deadlock detector for Java-like phasers, applicable also to barriers.

One feature of the original phaser that we have not addressed in this work is the single-statement **next** operation. This is a variant of **next** that takes a (non-blocking) statement as a parameter, to be executed exactly once after the relevant phasers are completely signaled and before any of the relevant tasks are released. There is an additional capability, SIGWAITNEXT > SIGWAIT, that is required in order to issue such an operation. Moreover, all tasks issuing a single-statement **next** on a given phaser must issue the same statement. The semantics of single-statement **next** can be simulated by advancing the phaser twice and designating one of the tasks to execute the statement in between.

Bounded phasers [76] support an additional parameter that imposes a limit on the difference between the highest signaled phase and the lowest observed phase. In producer-consumer patterns, such a bound on the phase difference means that tasks can safely share data using a buffer of limited size. Semantically, the **signal** operation of a bounded phaser may block. With a suitable type system, bounded phasers retain a deadlock-freedom guarantee [20]. It may be possible to unify the semantics of subphases with bounded phasers, for example, with different bounds applying at different subphase levels.

The hierarchical phaser [77] is an implementation of the ordinary phaser that is suitable when the number of parties is large. Instead of all parties competing on the same phaser object, multiple phasers are organized together as a tree to reduce contention. The splitting of a phaser into subphasers as defined by this technique is thus not a semantically meaningful generalization of the phaser, but merely an im-

plementation optimization. Our semantic change to phasers, supporting non-integer phase numbers, can also enable a reduction in contention on phasers, but in a way that actually has bearing on the synchronization dependences. It becomes possible, under our approach, to eliminate unnecessary signals and waits.

A fuzzy barrier separates the barrier arrival operation into its constituent signal and await sub-operations, allowing a task to perform some work after arriving instead of immediately blocking [40]. This feature, also known as a *split-phase*, is reflected in phasers: Even when the phaser-specific **wait** operation is forbidden, we still have the ability to **signal** specific phasers, perform subsequent work, and then invoke **next**. The complexity of the phaser semantics in using not one but two maps for storing the last signaled and last observed phase numbers is due to the split-phase feature. Without phaser-specific signals, the signal and the wait for any given phaser always occur together in **next**; the S and O maps in the semantics would always be in agreement in that case and could be conflated.

5.8 Conclusion

The phaser is a versatile synchronization primitive that comes equipped with a problematically restrictive deadlock-freedom constraints. We demonstrated its limitations by showing programs where a global **next** operation causes unwanted synchronization that ties together phasers which should be semantically separated. In very simple scenarios where this problem arises in regard to awaiting task termination, the anti-modular behavior of phasers can be mitigated using finish blocks. However, in more complex programs the same difficulties arise but do not coincide with task termination. Therefore a more powerful solution is required.

We formulated phaser semantics in such a way that a few small changes can dramatically increase its versatility. We generalized the phase number counter to a list of counters. Our solution permits multiple phases of one phaser to elapse within

the same time period as a single phase of another phase simply by targeting different counters in the phase numbers. We used a block-structured approach to organize code into different *subphase levels* so that the correct manipulation of phase numbers happens automatically in a composable way. We proved that our extended phaser semantics still enjoys the traditional deadlock-freedom guarantee.

Finally, we implemented our phaser extension and evaluated its performance on benchmark programs compared with the use of traditional phasers both with and without complying with the standard deadlock-freedom usage constraints imposed by Habanero phasers. The evaluation suggests that for certain classes of synchronization, the use of subphases can recover efficiency that is lost to the use of phasers under the deadlock-freedom constraints. In one case, the use of subphases even outperformed the baseline that disregarded the constraints, demonstrating that our phaser extension actually provides novel and advantageous expressivity in phasers, admitting more efficient solutions to complex phaser-based programming problems.

CHAPTER 6
CONCLUSION

Facilitating the writing of safe, understandable code that takes full advantage of available parallelism is an essential contribution in post-Moore application development. Logical deadlock bugs arise in settings that use simple, yet powerful parallel primitives. We have addressed deadlocks for three existing primitives by applying the principle that well-designed usage policies restrict parallel behaviors to structured deadlock-free patterns, that such policies are efficiently verifiable by a runtime, and that the restrictions do not impose undue limitations on synchronization expressivity.

A More Permissive Deadlock-Freedom Policy for Futures. Async-future parallelism is naturally deadlock-free if futures are not shared through data races. Since race detection is a difficult and expensive problem both statically and dynamically, a better solution to guaranteeing deadlock freedom is to use a policy consisting of a few simple rules that are easy to verify. We defined such a policy, called Transitive Joins, which decides whether it is legal for a task to await a given future based on a lowest common ancestor calculation in the tree of tasks. The rules of the policy follow intuitively from the semantics of asynchronous task closures. Unlike direct cycle detection, which may traverse every task in the worst case, the time complexity of our policy-based verifier is bounded by the task tree depth, which is typically small in practice. Transitive Joins is more permissive than prior work on deadlock freedom policies for futures, admitting a new class of deadlock-free programs. In fact, Transitive Joins is maximally permissive with respect to the information it uses to detect

deadlocks, and, moreover, this is less information than is required by prior work. Some programs that share futures among tasks through properly synchronized (non-racy) mechanisms which do not introduce blocking dependences are deadlock-free and are now recognized as such by Transitive Joins. As a runtime verifier, Transitive Joins is able to check awaits on futures against the policy with low overhead.

In the future, to eliminate the verification overhead entirely, it may be possible to develop a static analysis based on Transitive Joins. Even if the analysis is imprecise, it could still determine that some await instances are always safe and remove the call to the verifier in those cases. This is possible since each await is considered individually by the policy, which is stateless with respect to join history.

Ownership Semantics for Promises to Enable Deadlock Avoidance. Unlike futures, promises are not bound to specific tasks. This relaxation improves their versatility so that many blocking synchronization mechanisms can be expressed through promises. However, this relaxation also inhibits deadlock avoidance and detection. We proposed tracking the current owning task for each promise by annotating task-creation syntax in a manner that is familiar to users of data-driven futures and phasers, which require task registration. Imposing our intuitive and easy-to-verify ownership policy can improve the quality of promise-based code: each promise is owned by exactly one task until it is fulfilled and may be handed off to a newly spawned task. It is a runtime error for a task to terminate while holding an unfulfilled promise; this feature alone eliminates an entire class of deadlock-like bugs, the omitted set, wherein a task awaits a promise that will never be fulfilled. Once we are able to discuss promise ownership, it becomes possible to meaningfully define and identify promise deadlocks for the first time. We provided an algorithm which performs cycle detection over the ownership information to precisely reject exactly the cycle-forming awaits. We showed how to make this algorithm robust to weak

memory models, and we proved it to be correct and precise under minimal memory consistency assumptions.

Once again, a future consideration is whether ownership information and the ownership policy can be resolved statically. A natural approach to guaranteeing that every promise is owned by exactly one task at a time and is dispatched, either by being fulfilled or by being handed off to a new task, is to invoke a linear type system. Linear types need not be applied to the promises directly, but to an auxiliary token representing the obligation to fulfill. Such linear tokens may also be used to declare that certain promises have one and only one consumer. In that scenario it becomes possible to hand off promise obligations through promises or other mechanisms beyond the task-creation annotation.

Approximate Deadlock Avoidance Promises In using a policy-based deadlock verifier for futures, we reject some deadlock-free programs, but we argued that such programs exhibit bad style due to disorganized synchronization. We made the same case for promise-based code and advocated for an approximate cycle detection policy for promises. Building on the lowest common ancestors reasoning used in Transitive Joins, we let the children of the common ancestor stand in as representatives of two tasks in a blocking dependence. The rule to follow is that transitive chains of blocking dependences among the same set of siblings should be convex with respect to the sibling order, thereby precluding cycles. Because this policy localizes the relevant information to common ancestor nodes, the verification process can avoid traversing long chains of dependences, unlike a precise cycle detector. Moreover, we introduced a novel pseudo-synchronization primitive, the guard block, in conjunction with our policy. A guard block exempts all contained code from deadlock verification provided that it is permissible to await a promise specified in the guard header. The function of guards is then two-fold: decreasing verifier overhead by reducing the deadlock

problem to a small number of key dependences and safely performing policy-violating synchronization in delimited regions of code without actually breaking the verification guarantees.

One of the complexities of our approximate deadlock detection for promises that did not arise for futures is that the promise verifier necessarily maintains shared internal state. Two consequences are that the verifier is susceptible to contention and the validity of certain waits changes over time, subject to the nondeterministic internal state as governed by the waits issued throughout the rest of the program. It is highly preferable for a given wait to be either valid or invalid on its face, which is true in Transitive Joins, rather than depending on information not visible to the programmer. A future improvement to our policy for promises could require a parent task to declare the manner in which its descendant tasks will synchronize, to disambiguate which waits are valid and which are invalid up front rather than dynamically.

Subphase Numbering for Reducing Over-Synchronization. Complex group-based synchronization patterns can often be captured through the use of phasers. The existing approach to a deadlock-freedom guarantee for phasers imposes a constrained, coarse-grained interface that often results in unwanted synchronization dependences. Unrelated phasers become correlated, and tasks which should not interact with one another must be aware of each other's design. In the worst cases, a task may need to wastefully pump a phaser that it ought to be insulated from for a very specific number of iterations. Finish blocks can solve the design flaw only in limited scenarios where the issue coincides with task termination. We introduced the subphase primitive, which allows tiered grouping of phases. Subphase blocks insulate the interactions of some tasks and phasers from others, permitting modular code design. We showed empirically that subphases eliminate unnecessary synchronization that would otherwise be introduced by complying with the traditional deadlock-free approach. For some

programs, subphase blocks can even eliminate unnecessary synchronization that is not attributable to the deadlock-freedom constraints, showing that subphase blocks add fundamentally new expressivity to phasers. We proved that programs using subphase blocks still enjoy the traditional deadlock-freedom guarantee for phasers.

The user experience for subphases may be improved with an alternative syntax or adjustments to the underlying semantics. For example, the truncate-and-increment operation could be performed automatically upon exiting a subphase block, which simplifies some of the example programs. One could also allow phaser interactions at numerically specified subphase levels or the insertion of new subphase levels between existing ones. Finally, there may be some complex use cases where it is helpful to organize phasers not into linear levels but into a tree hierarchy.

* * *

In all three domains, futures, promises, and phasers, we explored opportunities for deadlock-freedom verification that are enabled through carefully designed policies and synchronization primitives. Transitive Joins reduces the false alarm rate for futures without modifying user code. Simple task annotations make deadlock detection possible for promises by providing an ownership framework for reasoning about cycles and omitted sets. Approximate cycle detection, together with the novel guard block, allows promise synchronization to be organized at a higher level of granularity for improved deadlock reasoning. Subphases allow for the decoupling of unrelated groups of tasks and phasers to eliminate unnecessary synchronization while preserving the deadlock-freedom guarantee for phasers. The design of task-parallel programming languages not only impacts the ease of expressing the desired computations but also governs the way we think about algorithms and the computational dependences within them. These policies and features advance the state of deadlock detection and deadlock-free code design in a principled manner that enables parallel code to be safer, better organized, and more efficient.

REFERENCES

[1] Google 2020. *Asynchronous Programming: Futures, Async, Await.* Google. Retrieved 8 October 2020 from https://dart.dev/codelabs/async-await

[2] Rahul Agarwal and Scott D. Stoller. 2006. Run-Time Detection of Potential Deadlocks for Programs with Locks, Semaphores, and Condition Variables. In *Proc. 2006 Worksh. Parallel and Distributed Systems: Testing and Debugging.* 51–60.

[3] Dongie Agnir. 2019. *Call exceptionOcurred in case of stream error.* Amazon Web Services. Retrieved 30 July 2020 from https://github.com/aws/aws-sdk-java-v2/commit/bfdd0d2063

[4] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele, Jr., and Sam Tobin-Hochstadt. 2008. *The Fortress Language Specification.* Sun Microsystems, Inc.

[5] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. 1989. I-Structures: Data Structures for Parallel Computing. *ACM Trans. Program. Lang. Syst.* 11, 4 (1989), 598–632.

[6] D.H. Bailey, E. Barszcz, J.T. Barton, D.S. Browning, R.L. Carter, L. Dagum, R.A. Fatoohi, P.O. Frederickson, T.A. Lasinski, R.S. Schreiber, H.D. Simon, V. Venkatakrishnan, and S.K. Weeratunga. 1991. The NAS Parallel Benchmarks. *Internat. J. Supercomputing Applic.* 5, 3 (1991), 63–73.

[7] Michael A. Bender and Martín Farach-Colton. 2004. The Level Ancestor Problem Simplified. *Theor. Comput. Sci.* 321, 1 (2004), 5–12.

[8] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proc. 17th Intl. Conf. Parallel Architectures and Compilation Techniques.* 72–81.

[9] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling Multithreaded Computations by Work Stealing. *J. ACM* 46, 5 (1999), 720–748.

[10] Rastislav Bodik, Rajiv Gupta, and Vivek Sarkar. 2000. ABCD: Eliminated Array Bounds Check on Demand. In *Proc. ACM SIGPLAN 2000 Conf. Programming Language Design and Implementation.* 321–333.

[11] Gérard Boudol. 2009. A Deadlock-Free Semantics for Shared Memory Concurrency. In *Proc. 6th Intl. Coll. Theoretical Aspects of Computing*. 140–154.

[12] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. 2002. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *Proc. 17th ACM SIGPLAN Conf. Object Oriented Programming, Systems, Languages, and Applications*. 211–230.

[13] Jeremy D. Buhler, Kunal Agrawal, Peng Li, and Roger D. Chamberlain. 2012. Efficient Deadlock Avoidance for Streaming Computation with Filtering. In *Proc. 17th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*. 235–246.

[14] John Burkardt. 2016. *HEAT_MPI: Solve the 1D Time Dependent Heat Equation using MPI*. Florida State University. Retrieved 13 August 2020 from https://people.sc.fsu.edu/~jburkardt/cpp_src/heat_mpi/heat_mpi.html

[15] Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. 2011. Habanero-Java: The New Adventures of Old X10. In *Proc. 9th Intl. Conf. Principles and Practices of Programming in Java*. 51–61.

[16] Bradford L. Chamberlain, David Callahan, and Hans P. Zima. 2007. Parallel Programmability and the Chapel Language. *Intl. J. High Perf. Comput.* 21, 3 (2007), 291–312.

[17] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. X10: An Object-oriented Approach to Non-uniform Cluster Computing. In *Proc. 20th ACM SIGPLAN Conf. Object Oriented Programming, Systems, Languages, and Applications*. 519–538.

[18] Edward G. Coffman, Michael J. Elphick, and Arie Shoshani. 1971. System Deadlocks. *ACM Comput. Surv.* 3, 2 (1971), 67–78.

[19] Tiago Cogumbreiro, Raymond Hu, Fransisco Martins, and Nobuko Yoshida. 2015. Dynamic Deadlock Verification for General Barrier Synchronisation. In *Proc. 20th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*. 150–160.

[20] Tiago Cogumbreiro, Fransisco Martins, and Vasco T. Vasconcelos. 2013. Coordinating Phased Activities while Maintaining Progress. In *15th Intl. Conf. Coordination Models and Languages*. 31–44.

[21] Tiago Cogumbreiro, Jun Shirako, and Vivek Sarkar. 2017. Formalization of Habanero phasers using Coq. *Journal of Logical and Algebraic Methods in Programming* 90 (2017), 50–60.

[22] Tiago Cogumbreiro, Rishi Surendran, Fransisco Martins, Vivek Sarkar, Vasco T. Vasconcelos, and Max Grossman. 2017. Deadlock Avoidance in Parallel Programs with Futures: Why parallel tasks should not wait for strangers. *Proc. ACM Programming Languages* OOPSLA, Article 103 (2017), 26 pages.

[23] Thomas M. Conte, Erik P. DeBenedictis, Paola A. Gargini, and Elie Track. 2017. Rebooting Computing: The Road Ahead. *IEEE Computer* 50, 1 (2017), 20–29.

[24] Alex Crichton. 2020. *futures – Rust.* Retrieved 24 August 2020 from https://docs.rs/futures

[25] Alex Crichton. 2020. *futures::channel – Rust.* Retrieved 30 July 2020 from https://docs.rs/futures/0.3.5/futures/channel

[26] Edsger W. Dijkstra. 1968. Letters to the Editor: Go To Statement Considered Harmful. *Commun. ACM* 11, 3 (1968), 147–148.

[27] Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Eduard Ayguade. 2009. Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP. In *Proc. 2009 Intl. Conf. Parallel Processing*. 124–131.

[28] Kiko Fernandez-Reyes, Dave Clarke, Elias Castegren, and Huu-Phuc Vo. 2018. Forward to a Promising Future. In *20th Intl. Conf. Coordination Models and Languages*. 162–180.

[29] John G.F. Francis. 1961. The QR Transformation: A Unitary Analogue to the LR Transformation—Part 1. *Comput. J.* 4, 3 (1961), 265–271.

[30] Rhys S. Francis and Linda J.H. Pannan. 1992. A Parallel Partition for Enhanced Parallel QuickSort. *Parallel Comput.* 18, 5 (1992), 543–550.

[31] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The Implementation of the Cilk-5 Multithreaded Language. In *Proc. ACM SIGPLAN 1998 Conf. Programming Language Design and Implementation*. 212–223.

[32] Donald Fussell, Zvi M. Kedem, and Abraham Silberschatz. 1981. Deadlock Removal Using Partial Rollback in Database Systems. In *1981 ACM SIGMOD Intl. Conf. Management of Data*. 65–73.

[33] Narain H. Gehani and William D. Roome. 1988. Rendezvous Facilities: Concurrent C and the Ada Language. *IEEE Trans. Soft. Eng.* 14, 11 (1988), 1546–1553.

[34] Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically Rigorous Java Performance Evaluation. In *Proc. 22th ACM SIGPLAN Conf. Object Oriented Programming, Systems, Languages, and Applications*. 57–76.

[35] Prodromos Gerakios, Nikolaos Papaspyrou, Konstantinos Sagonas, and Panagiotis Vekris. 2011. Dynamic Deadlock Avoidance in Systems Code Using Statically Inferred Effects. In *Proc. 6th Worksh. Programming Languages and Operating Systems*. Article 5, 5 pages.

[36] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, Doug Lea, and D. Holmes. 2006. *Java Concurrency in Practice*. Pearson Education, London, England.

[37] Google. 2014. *Go 1.3 Release Notes*. Retrieved 30 July 2020 from https://golang.org/doc/go1.3

[38] James Gosling, Bill Joy, Guy L. Steele, Jr., Gilad Bracha, Alex Buckley, Lorna A. Smith, and Gavin Bierman. 2020. *The Java Language Specification*. Oracle America, Inc. https://docs.oracle.com/javase/specs/jls/se14/jls14.pdf

[39] Yi Guo, Rajkishore Barik, Raghavan Raman, and Vivek Sarkar. 2009. Work-first and Help-first Scheduling Policies for Async-finish Task Parallelism. In *Proc. 2009 IEEE Intl. Symp. Parallel and Distributed Processing*. 1–12.

[40] Rajiv Gupta. 1989. The Fuzzy Barrier: A Mechanism for High Speed Synchronization of Processors. In *Proc. Third Intl. Conf. Architectural Support for Programming Languages and Operating Systems*. 54–63.

[41] Habanero Extreme Scale Software Research Lab. 2020. *Habanero-C Library*. https://github.com/habanero-rice/hclib

[42] Philipp Haller, Aleksandar Prokopec, Heather Miller, Viktor Klang, Roland Kuhn, and Vojin Jovanovic. 2012. *Futures and Promises*. École polytechnique fédéral de Lausanne. Retrieved 24 August 2020 from https://docs.scala-lang.org/overviews/core/futures.html

[43] Robert H. Halstead, Jr. 1985. MULTILISP: A Language for Concurrent Symbolic Computation. *ACM Trans. Program. Lang. Syst.* 7, 4 (1985), 501–538.

[44] Kathleen E. Hamilton, Catherine D. Schuman, Steven R. Young, Ryan S. Bennink, Neena Imam, and Travis S. Humble. 2020. Accelerating Scientific Computing in the Post-Moore's Era. *ACM Trans. Par. Comput.* 7, 1, Article 6 (2020).

[45] Dov Harel and Robert Endre Tarjan. 1984. Fast Algorithms for Finding Nearest Common Ancestors. *SIAM J. Comput.* 13, 2 (1984), 338–355.

[46] Tobias Hilbrich, Bronis R. de Supinski, Martin Schulz, and Matthias S. Müller. 2009. A Graph Based Approach for MPI Deadlock Detection. In *Proc. 23rd Intl. Conf. Supercomputing*. 296–305.

[47] Tobias Hilbrich, Joachim Protze, Martin Schulz, Bronis R. de Supinski, and Matthias S. Müller. 2012. MPI Runtime Error Detection with MUST: Advances in Deadlock Detection. In *Proc. Intl. Conf. High Performance Computing, Networking, Storage and Analysis*. Article 30, 11 pages.

[48] Oliver Hsu. 2019. *S3: FileAsyncResponseTransformer future does not complete when checksum error occurs*. Amazon Web Services. Retrieved 30 July 2020 from https://github.com/aws/aws-sdk-java-v2/issues/1279

[49] Shams Imam and Vivek Sarkar. 2014. Cooperative Scheduling of Parallel Tasks with General Synchronization Patterns. In *Proc. 28th European Conf. Object-Oriented Programming*. 618–643.

[50] Shams Imam and Vivek Sarkar. 2014. Habanero-Java Library: A Java 8 Framework for Multicore Programming. In *Proc. 2014 Intl. Conf. Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. 75–86.

[51] Intel 2020. *Intel Threading Building Blocks Developer Guide*. Intel.

[52] S. Sreekaanth Isloor and T. Anthony Marsland. 1980. The Deadlock Problem: An Overview. *IEEE Computer* 13, 9 (1980), 58–78.

[53] ISO. 2017. *ISO/IEC 14882:2017: Programming Languages — C++*. International Organization for Standardization, Geneva, Switzerland.

[54] Naoki Kobayashi. 1998. A Partially Deadlock-Free Typed Process Calculus. *ACM Trans. Program. Lang. Syst.* 20, 2 (1998), 436–482.

[55] Bettina Krammer, Tobias Hilbrich, Valentin Himmler, Blasius Czink, Kiril Dichev, and Matthias S. Müller. 2008. MPI Correctness Checking with Marmot. In *Proc. 2nd Intl. Worksh. Parallel Tools for High Performance Computing*. 61–78.

[56] Bettina Krammer, Matthias S. Müller, and Michael M. Resch. 2004. MPI Application Development Using the Analysis Tool MARMOT. In *Proc. Intl. Conf. Computational Science*. 464–471.

[57] Robert Lee and Jens Palsberg. 2010. Featherweight X10: A Core Calculus for Async-Finish Parallelism. *SIGPLAN Not.* 45, 5 (2010), 25–36.

[58] Matthew C. Loring, Mark Marron, and Daan Leijen. 2017. Semantics of Asynchronous JavaScript. In *Proc. 13th ACM SIGPLAN Intl. Symp. Dynamic Languages*. 51–62.

[59] Glenn Luecke, Hua Chen, James Coyle, Jim Hoekstra, Marine Kraeva, and Yan Zou. 2003. MPI-CHECK: A Tool for Checking Fortran 90 MPI Programs. *Concurrency and Computation: Practice and Experience* 15, 2 (2003), 93–100.

[60] Mayur Naik, Chang-Seo Park, Kousik Sen, and David Gay. 2009. Effective Static Deadlock Detection. In *Proc. 31st Intl. Conf. Software Engineering*. 386–396.

[61] Varun Nandi. 2019. *Don't call onComplete after onError in ChecksumValidatingSubscriber#onComplete method which results in NPE.* Amazon Web Services. Retrieved 30 July 2020 from https://github.com/aws/aws-sdk-java-v2/commit/eaecf99a02

[62] Armand Navabi, Xiangyu Zhang, and Suresh Jagannathan. 2008. Quasi-static Scheduling for Safe Futures. In *Proc. 13th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*. 23–32.

[63] Mozilla Developer Network. 2020. *Promise – JavaScript — MDN.* Retrieved 5 August 2020 from https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

[64] Nicholas Ng and Nobuko Yoshida. 2016. Static Deadlock Detection for Concurrent Go by Global Session Graph Synthesis. In *Proc. 25th Intl. Conf. Compiler Construction*. 174–184.

[65] Joachim Niehren, Jan Schwinghammer, and Gert Smolka. 2005. A Concurrent Lambda Calculus with Futures. In *Intl. Worksh. Frontiers of Combining Systems*. 338–356.

[66] OpenMP Architecture Review Board 2018. *OpenMP Application Programming Interface*. OpenMP Architecture Review Board.

[67] Oracle. 2020. *CompletableFuture (Java SE 14 & JDK 14).* Retrieved 30 July 2020 from https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/util/concurrent/CompletableFuture.html

[68] Oracle. 2020. *CyclicBarrier (Java SE 14 & JDK 14).* Retrieved 14 September 2020 from https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/util/concurrent/CyclicBarrier.html

[69] Oracle. 2020. *Phaser (Java SE 14 & JDK 14).* Retrieved 24 August 2020 from https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/util/concurrent/Phaser.html

[70] Alex Pothen and Padma Raghavan. 1989. Distributed Orthogonal Factorization. In *Proc. Third Conf. Hypercube Concurrent Computers and Applications*. 1610–1620.

[71] Spiridon A. Reveliotis, Mark A. Lawley, and Placid M. Ferreira. 1997. Polynomial-Complexity Deadlock Avoidance Policies for Sequential Resource Allocation Systems. *IEEE Trans. Autom. Control* 42, 10 (1997), 1344–1357.

[72] Rust Lang. 2020. *The Rust Programming Language*. Retrieved 12 August 2020 from https://doc.rust-lang.org/1.8.0/book/index.html

[73] Jun Shirako, Vincent Cavé, Jisheng Zhao, and Vivek Sarkar. 2013. Finish Accumulators: An Efficient Reduction Construct for Dynamic Task Parallelism. In *26th Intl. Worksh. Languages and Compilers for Parallel Computing*. 264–265.

[74] Jun Shirako, David M. Peixotto, Vivek Sarkar, and William N. Scherer, III. 2008. Phasers: A Unified Deadlock-Free Construct for Collective and Point-to-Point Synchronization. In *Proc. 22rd Intl. Conf. Supercomputing*. 277–288.

[75] Jun Shirako, David M. Peixotto, Vivek Sarkar, and William N. Scherer, III. 2009. Phaser Accumulators: A New Reduction Construct for Dynamic Parallelism. In *Proc. 2009 IEEE Intl. Symp. Parallel and Distributed Processing*. 1–12.

[76] Jun Shirako, David M. Peixotto, Dragoş-Dumitru Sbîrlea, and Vivek Sarkar. 2011. Phaser Beams: Integrating Stream Parallelism with Task Parallelism. In *Proc. ACM SIGPLAN X10 Worksh.*

[77] Jun Shirako and Vivek Sarkar. 2010. Hierarchical Phasers for Scalable Synchronization and Reductions in Dynamic Parallelism. In *Proc. 2010 IEEE Intl. Symp. Parallel and Distributed Processing*. 1–12.

[78] Lorna A. Smith, J. Mark Bull, and Jan Obdrzálek. 2001. A Parallel Java Grande Benchmark Suite. In *Proc. 2001ACM/IEEE Conf. Supercomputing*. Article 8, 10 pages.

[79] Sağnak Taşırlar and Vivek Sarkar. 2011. Data-Driven Tasks and Their Implementation. In *2011 Intl. Conf. Parallel Processing*. 652–661.

[80] Vasco T. Vasconcelos, Fransisco Martins, and Tiago Cogumbreiro. 2010. Type Inference for Deadlock Detection in a Multithreaded Polymorphic Typed Assembly Language. In *Electronic Proceedings in Theoretical Computer Science (Proc. 2nd Intl. Worksh. Programming Language Approaches to Concurrency and Communication-cEntric Software, Vol. 17)*, Alastair R. Beresford and Simon Gay (Eds.). 95–109.

[81] Philippe Virouleau, Pierrick Brunet, François Broquedis, Nathalie Furmento, Samuel Thibault, Olivier Aumage, and Thierry Gautier. 2014. Evaluation of OpenMP Dependent Tasks with the KASTORS Benchmark Suite. In *Intl. Worksh. OpenMP*. 16–29.

[82] Anh Vo, Ganesh Gopalakrishnan, Robert M. Kirby, Bronis R. de Supinski, Martin Schulz, and Greg Bronevetsky. 2011. Large Scale Verification of MPI Programs Using Lamport Clocks with Lazy Update. In *Proc. 2011 Intl. Conf. Parallel Architectures and Compilation Techniques*. 330–339.

[83] Caleb Voss, Tiago Cogumbreiro, and Vivek Sarkar. 2019. Transitive Joins: A Sound and Efficient Online Deadlock-Avoidance Policy. In *Proc. 24th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*. 378–390.

[84] Caleb Voss and Vivek Sarkar. 2021. An Ownership Policy and Deadlock Detector for Promises. In *Proc. 26th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*. To Appear.

[85] Aaron Weeden. 2012. *Parallelization: Conway's Game of Life*. The Shodor Education Foundation. Retrieved 13 August 2020 from http://www.shodor.org/petascale/materials/UPModules/GameOfLife

[86] Adam Welc, Suresh Jagannathan, and Antony Hosking. 2005. Safe Futures for Java. In *Proc. 20th ACM SIGPLAN Conf. Object Oriented Programming, Systems, Languages, and Applications*. 439–453.

[87] Amy Williams, William Thies, and Michael D. Ernst. 2005. Static Deadlock Detection for Java Libraries. In *Proc. 19th European Conf. Object-Oriented Programming*. 602–629.

VITA

Caleb Voss is a native of Lexington, KY, arriving in Georgia by way of Texas and, briefly, California. Voss is an alumnus of Rice University (B.A. '16, *magna cum laude*), where he was a Trustee Distinguished Scholar and Century Scholar, receiving the James S. Waters Creativity in Research award and the Rice Engineering Alumni award for Outstanding Research Excellence. At the Georgia Institute of Technology (Ph.D. '20) he was a National Science Foundation Graduate Research Fellow and President's Fellow, receiving the College of Computing's award for Outstanding Graduate Teaching Assistant. In addition to his dissertation advisor, Prof. Vivek Sarkar (Georgia Tech), Voss has also studied under the mentorship of Prof. Lydia Kavraki (Rice) and Dr. Mark Moll (formerly Rice), Prof. James Griffioen (University of Kentucky), and Dr. William Harris (formerly Georgia Tech). His research interests have included parallel programming techniques, language design, formal program analysis, and robotic motion planning algorithms. Voss has experience teaching programming language theory at both the undergraduate and graduate levels.